

Moving from Python to C++

Writing better object-oriented programs



James W Cooper

Moving from Python to C++

James W Cooper



Rev 3-15-2024

Fairfield Easton Press
2024

Other books by James W Cooper

Python Programming w/ Design Patterns, ISBN:0-13-757993-4

Flameout, ISBN: 9781706406358

Food Myths Debunked, ISBN:978-1502386007

Where to Dine in Nantucket, ISBN: 978-1-304-19277-6

The Hollow, ISBN: 5-800051-790432

Introduction to Cooking for Graduate Students

C# Design Patterns: A Tutorial, ISBN:0-201-84453-2

Visual Basic Design Patterns, ISBN: 0-201-70265-7

Java Design Patterns: A Tutorial, ISBN: 0-201-48539-7

Principles of Object-Oriented Programming in Java, 1-56604-530-4

The Visual Basic Programmer's Guide to Java, 1-56604-527-4

Object Oriented Programming in Visual Basic, 1-880935-49-x

A Jump Start Course in C++ Programming, 0-471-03171-2

Visual BASIC for DOS, ISBN: 0-471-59772-4

Writing Scientific Programs Under OS/2, ISBN:0-471-51928-6

Microsoft QuickBASIC for Scientists, ISBN: 0-471-61301-0

The Laboratory Microcomputer, ISBN: 0-471-81036-3

The Minicomputer in the Laboratory, 2nd ed 0-471-09012-3

Introduction to Pascal for Scientists, ISBN: 0-471-08785-8

Spectroscopic Techniques for Organic Chemists 0-471-05166-7

The Minicomputer in the Laboratory, ISBN: 0-471-01883-X

Copyright © 2024 by James W. Cooper

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review or scholarly journal.

First Printing 2024

Fairfield Easton Press

48 Old Driftway

Wilton, CT 06897

www.jameswcooper.com

Contents

Introduction	17
How to use GitHub	18
Reference	19
Book organization	21
PART I – Learning C++	21
PART II -Application Development	22
Part III – Design patterns	22
1. Basic C++ Syntax	23
Variables	23
Statements	23
Declaring variable types	24
Data types in C++	25
Character Constants	26
The size_t type	27
Arithmetic Operations	27
Converting between numeric types	28
Arithmetic shortcuts	28
A complete program	29
Bitwise operators	30
Types of Integers	31
Positions of braces	31
The auto keyword	32
Example code	32
References	32
2. C++ Development Systems	33

Visual Studio	34
CodeBlocks	34
CLion.....	35
Other IDE systems.....	36
References	36
3. Input and Output.....	37
Output using cout	37
Input using cin	38
Reading in whole lines	38
The std namespace.....	39
Formatting in Python.....	39
Formatting in C++	40
Symbols in format function	42
Alignment and fill formatting.....	43
Error Handling in formatting.....	44
File handling.....	45
Binary files	46
Reading a binary file	47
Example files on GitHub.....	48
References	48
4. Loops, Arrays and Strings	49
Two-dimensional arrays	50
The for loop.....	51
The range based for loop	51
The while loops	52
Vectors	52

Vector Methods.....	54
Strings.....	55
Reversing a string.....	56
String methods.....	57
Changing string case.....	58
Converting numbers to strings and vice-versa	58
Example Code on GitHub	59
5. Making Decisions.....	60
Elif is “else if”	60
Combining Conditions.....	61
The Most Common Mistake.....	61
Comparing strings	62
A ticketing program.....	62
The switch statement	63
Break and continue	65
The ornery ternary operator.....	67
Example programs on GitHub.....	68
6. Functions	69
Function order	70
Polymorphism in functions.....	71
Function prototypes.....	71
Passing arguments to functions	72
Default arguments	73
Using constant declarations	74
Example programs.....	74
7. Using Pointers	75

Arrays and pointers.....	76
Calling functions	77
Functions and Arrays.....	78
C strings and pointers	80
Example code on GitHub	81
8. Sets, tuples and maps.....	83
Sets	83
Merging sets	84
Tuples	85
Maps and Dictionaries.....	87
Example programs in GitHub.....	88
9. Classes and OOP	89
A Rectangle class.....	89
Inheritance	92
More useful classes.....	93
Deriving new classes	96
Public, protected and private inheritance	97
Classes within a class	98
Classes and headers	100
Using Headers	103
The main program	104
Summary of headers	105
Multiple Inheritance	105
Polymorphism	107
Virtual Functions	108
Pure Virtual Functions.....	109

Static class members.....	111
Friend declarations	112
Constant classes.....	112
Example Programs.....	113
10. Pointers and Memory	115
Classes and destructors.....	116
When is the destructor called?.....	117
Other uses for destructors	118
Smart Pointers	118
Example Code on GitHub	120
11. Using linked lists	121
Definitions	121
Creating the list	123
Traversing the list	123
The reverse iterator.....	125
Inserting a new cell in the chain	125
The copy constructor	126
Is this trip really necessary?	127
Deleting the copy constructor.....	128
Summary	128
Example Code	128
12. Templates.....	129
Template functions	129
Class templates	130
Class templates of classes.....	132
References	135

Example Code	135
13. Creating user interfaces	137
A wxPython example.....	138
Strings in wxWidgets.....	140
Writing basic wxWidgets code	140
Sizers	142
Include Files	142
The Box Sizer.....	142
Splitting up the main app.....	143
More on labels	144
Entry fields and buttons.....	145
Events in wxWidgets	147
Adding two numbers together	149
The GridBag Sizer	149
The Add and Clear buttons	151
Command Buttons	152
Menus	154
Shortcuts and accelerators	156
Radio or check menuitems	157
Binding MenuItem's	157
Dialog Boxes	158
The File Dialog.....	159
Installing wxWidgets.....	159
Example Programs on GitHub.....	160
References	161
14. Choices and Listboxes	163

RadioButtons	163
Reading the Radio buttons.....	164
Responding to RadioButton clicks	165
Finding the calling object	167
ListBoxes.....	168
CheckListBoxes.....	170
The StateLister Application.....	171
Using a Mediator Class	172
The ComboBox	174
Checkboxes	175
Checkbox styles.....	177
Displaying tables in a grid.....	178
Selecting Grid regions	180
Other wxGrid features	182
The Tree widget.....	182
Moving on	183
Example programs on GitHub.....	183
Part II- Application Development.....	185
15. The Armadillo Math Library	187
Overview of Armadillo Classes.....	187
Matrices	188
Matrix methods.....	190
Matrix functions	190
Decompositions, Inverses and Equation Solvers.....	191
Signal and Image Processing.....	191
Matrix transpose	191

The transpose.....	192
The Fast Fourier transform.....	193
Curve fitting	195
Installing and running Armadillo programs.....	197
Running the example program	197
Running new armadillo programs	198
Example programs on GitHub.....	198
References	198
16. Plotting in C++	201
Plotting using DrawLines	203
SciPlot	205
ROOT	207
The ROOT interpreter	208
Writing C++ code for ROOT.....	211
Writing ROOT code for a C compiler	212
Error bars	213
Plotting multiple lines in ROOT.....	214
Example programs on GitHub.....	216
References	216
17. Databases in C++.....	219
SQLite	221
Downloading SQLite.....	221
SQLiteStudio	222
Programming SQLite in C++	223
Compiling using Visual Studio.....	223
Example C++ code to connect to SQLite.....	227

Building a database class structure.....	229
The Query object.....	230
The Results class.....	232
Using the SQLite classes.....	232
Database Tables.....	233
Adding rows to a Table.....	236
Prepared Queries.....	238
Summary.....	242
Example programs on GitHub.....	242
References.....	242
18. Using the MySQL database.....	243
Installing MySQL.....	243
Writing C++ to connect to MySQL.....	244
Debugging libraries for Connector C++.....	248
Creating C++ classes to connect to MySQL.....	248
Numeric types in MySQL.....	251
MySQL Query results.....	251
Why does printing out a Value object work?.....	252
Creating the MySQL groceries database.....	253
Prepared Queries in MySQL.....	254
Table functions in Connector C++.....	256
Other approaches to prepared queries.....	257
Example programs on GitHub.....	258
Summary.....	258
References.....	258
19. Namespaces and Modules.....	261

Modules	263
The module descriptor file.....	264
Make the descriptor file an include file	265
Combining namespaces and modules.....	265
Example code on GitHub	267
References	267
Part III -Design Patterns	269
Notes on Object Oriented Approaches	269
Commonly used Patterns.....	271
References	271
20. Factory Patterns	273
The Simple Factory Pattern.....	273
The Factory Method Pattern.....	276
Example programs on GitHub.....	282
21. The Abstract Factory Pattern	284
A GardenMaker Factory.....	284
The Plant class.....	285
A Garden class.....	285
How the GUI works.....	287
Example program In GitHub	289
22. Adapters.....	290
Moving Data between Lists.....	290
The Grid Adapter code	294
Class Adapters	296
The GridAdapter class.....	298
Finding the current row	300

Object adapters and class adapters	301
Example Code on GitHub	301
23. The Bridge pattern.....	302
The Bridge	304
The VisLists.....	305
How to set up the Bridge	307
Other VisLists	308
Summary	310
Example programs on GitHub.....	310
References	310
Index.....	312

Introduction

This book teaches you how to write programs in C++ and contrasts them with programs you might have written in Python. Since the syntax of the two languages are pretty similar, this should be a pretty easy transition. We also discuss libraries that you can use in place of Python's tkinter, Numpy and Matplotlib.

Why should you take up C++? Well, since C++ is a compiled language, that compiled code will run a lot faster. And you can compile your program to run on several different platforms without requiring the user to install Python or any other compiler.

The single major difference in C++ is that you enclose blocks of code in *braces* (`{ }`) rather than simply indenting. This eliminates those annoying Python “indentation error” messages that can sometimes be hard to correct.

The other major difference is that you must end every statement with a semicolon.

C++ is a strongly typed language and requires that you declare a type for every variable. Thus, you can't accidentally use the same variable name to represent different kinds of numbers or strings as you can in Python.

But since the syntax of C++ and Python are very similar, you will be able to write C++ code right away. In fact, you will recognize our early examples as being ones you could easily have written in Python.

Years ago, when Dick Lam and I wrote our first book on C++ [1], it was still a pretty new concept, and we wrote the book using examples right along as we became familiar with the language.

Today, C++ has grown enormously from those early days, and not only has a lot of neat new tricks, but it has also grown closer to Python. So, you will be pretty comfortable in C++ as we go along.

C++ is a satisfying experience because your code will be clearly structured and, of course, it will run a lot faster than the same code in Python.

We start at the beginning and take you through the language, adding on features in each chapter. And you will find the code for every example in the GitHub repository.

While you can write C++ on any platform and using any number of tools, we will concentrate on creating C++ programs on Windows using the free Community edition of Microsoft Visual Studio. But everything we write will run on all major platforms.

You might be wondering if with all the AI systems now available whether you still need to write your own programs at all. Of course, the answer is YES! You can use AI tools like CHAT-GPT to help you find how to write programs and provide you with examples, but you will get the best uses of those AI tools if you know how to ask the right questions. And those questions come from understanding the fundamentals of C++ programming. That is the objective of this book.

How to use GitHub

All of the example programs in this book are available for you to download from GitHub. Look at

`jwcnmr/jameswcooper/PyCpp`

In case you are unfamiliar with GitHub, it is a free software repository managed by Microsoft for sharing code; anyone can use it.

To get started, go to [GitHub.com](https://github.com) and click on Sign Up. You will need to create a user ID and a password and submit an Email

address for verification. Then you can search for any code repository (such as jameswcooper) and download any code you want. There is also a complete manual on that website. The complete path to the examples in this book is

<https://github.com/jwcnmr/jameswcooper/tree/main/PyCpp>.

If you are downloading a multifile project with both cpp and include (.h) files, then when you create your Visual Studio project, you should click on Header Files in the Solution Explorer and add all the include files, and then click on Source Files and add the .cpp files.

Reference

1. James W Cooper and Richard B Lam, *A Jump Start Course in C++ Programming*, New York: Wiley-Interscience, 1994.

Book organization

This book is divided into three major sections: Learning C++, Application Development and Design Patterns. While the chapters are really a continuum, the latter chapters take up a number of external packages you will want to use that are analogous to ones in Python. The Design Pattern section shows you useful techniques for writing more sophisticated programs.

PART I – Learning C++

- Chapter 1 introduces you to the syntax of the basic C++ language. Indentation is no longer required, but the programming tools usually help you choose an indentation style.
- Chapter 2 summarizes a number of the most popular Integrated Development Environments (IDEs), and makes some recommendations.
- Chapter 3 shows you how to write the equivalent of **print** and **input** statements, and how to read and write data from files.
- Chapter 4 takes you through arrays and more or less the same sort of looping statements you learned in Python.
- Chapter 5 shows you how you can make decision in C++, and you won't be too surprised that they also are pretty similar to Python and other languages.
- Chapter 6 introduces functions so you can group code into useful units.
- Chapter 7 introduces pointers: really the first new concept to you if you started in Python. They make it a lot easier to keep from copying data all over the place between functions
- Chapter 8 introduces sets, tuples and maps: all of which should seem familiar to you.
- Chapter 9 brings you to classes, objects object-oriented programming, which is pretty easy in C++, too.

- Chapter 10 explains how you can reserve and release memory, how you create pointers to refer to it.

PART II-Application Development

- Chapter 11 shows you how you can use pointer to create linked lists
- Chapter 12 explains how templates expand the power of C++ and why you are using them all the time without knowing it.
- Chapter 13 shows you how to create C++ program with a graphical user interface, or GUI.
- Chapter 14 continues from Chapter 13 explaining Listboxes and choice boxes in your GUI.
- Chapter 15 introduces the Armadillo math library
- Chapter 16 introduces several plotting libraries you can use.
- Chapter 17 introduces databases and SQLite
- Chapter 18 shows you how you can build the same interface to MySQL, an industrial strength client server database.
- Chapter 19 explains how you can put your code into modules.

Part III – Design patterns

- Chapters 20-23 summarize a few important Design Patterns used in creating more significant programs.
 - Chapter 20 summarizes the Simple Factory and Factory Method patterns.
 - Chapter 21 illustrates the Abstract Factory Pattern
 - Chapter 22 shows how to use the Adapter Pattern
 - Chapter 23 shows you how to use Bridge Pattern.

1. Basic C++ Syntax

If you know Python, you are a long way towards learning C++ already. They have similar syntax, functions and a class structure you will understand pretty quickly. So, in this introductory chapter, we'll concentrate on the differences between Python and C++.

Of course, the main difference is that C++ runs much faster since it is compiled directly into machine code. And C++ has a great deal more flexibility in the ways you can build programs. But the syntax is strikingly similar.

Variables

Variable names can be made up of upper and lowercase characters, along with numbers and underscores. And like Python, variable names can start with an underscore, although in C++ it doesn't signify any special properties. Like Python, case is significant, so `Apples` and `apples` are different variables. And like Python, variable names cannot contain spaces or any other special characters.

Statements

The layout of code in C++ is very flexible. You can write one or more statements on a single line or spanning multiple lines: whichever is clearest to the reader. You terminate states in C++ with a *semicolon* character rather than the newline character Python uses. There is no requirement for indenting blocks of code, but it does make programs more readable.

```
apples = 5;  
oranges = 5;
```

Or, you could write the same code on a single line:

```
apples = 5; oranges = 5;
```

And like Python, if the variables are to have the same value, you could also write

```
apples = oranges = 5;
```

Spacing between operators is optional, just as in Python, so you could (inadvisably) write:

```
apples=oranges=5;
```

Declaring variable types

If you were to put these two statements into a little C++ program, however, they wouldn't actually work! We've left out the most significant difference between Python and C++. The C++ language is *strongly typed* and you must declare the type of every variable when you first use it. So, the complete syntax for declaring these variables is

```
int apples = 5;  
int oranges = 5;
```

The convention in C++ is to declare the variable right where you use it, but you could also break that up into two declarations like this:

```
int apples, oranges;  
  
// and later:  
apples = 5;  
oranges = 5;
```

Note that here we have just introduced the single line comment, It starts with two slashes and continues to the end of the line. You can also create multiple line comments using the `/*` and `*/` delimiters:


```
/* we declare apples and oranges here,
   and pass the values later */
```

```
int apples, oranges;
```

The whole point of declaring the type of each variable is so the compiler can generate the most efficient code for that data type. Thus, C++ is a *strongly typed* language, where each variable must have a type declared. By contrast, Python is a weakly typed language where data types are resolved at run time, not compile time.

In Python, you could write

```
apple = "fruit" #and later write
apple = 5
```

and both would be correct. The type of **apple** would simply change for the new declaration. Doing this in C++:

```
string apple = "fruit";
//and later
apple = 5; // would lead to a compiler error message
```

Note that when we create strings of characters, they are enclosed in *double* quotes(""). The single quote (') is reserved for characters:

```
char c = 'a';
```

and can only hold a single character.

Data types in C++

The data types you use in C++ are pretty similar to those in Python as shown in Table 1-1.

int	Integer 2 or 4 bytes		
float	4 byte floating point	7 digits of precision	10^{-126} to 10^{127}
double	8 byte floating point	15 digits of precision	10^{-1023} to 10^{1024}
char	1 byte	one character	Use single quotes 'a'
bool	Boolean, 1 byte	true or false	
string	array of characters		Use double quotes "apple"

Table 1-1 - Data types in C++

Python's **float** type is equivalent to C++'s **double** type: Python does not have a 4 byte floating point type. In fact, these days, most C++ programmers use the **double** type exclusively, since memory is seldom a limitation, and the greater precision is useful.

Character Constants

C++ follows the C convention that the “whitespace characters” can be represented by preceding special characters with a backslash. Since the backslash itself is thus a special character, it can be represented by using a double backslash.

'\n'	newline (line feed)
'\r'	carriage return
'\t'	tab character
'\b'	backspace
'\f'	form feed
'\0'	null character
'\"'	double quote
'\''	single quote
'\\'	backslash

The `size_t` type

You will often see for loops using an index variable of type **`size_t`** instead of type **`int`**. The `size_t` type is an unsigned integer long enough to hold the largest number that the **`sizeof`** function can return, and in modern systems it is usually an unsigned long int, usually 64-bit. However, this can vary with the platform and `size_t` is more general for manipulating indices that could sometimes become very large. We will use it beginning in Chapter 6.

Arithmetic Operations

Like most languages, you have the choice of the usual operations:

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo (the remainder after division)

Unlike Python, C++ does not promote integers to floats when you specify division. So

```
9 / 4
```

does not result in 2.25, but just 2. The remainder is discarded. If you want the integer remainder you can get it from the modulo operator:

```
9 % 4
```

will result in a result of 1.

Doubles and floats do not convert to integers, so

```
5.0 / 2
```

will give 2.5.

Converting between numeric types

You can always convert from a narrower type to a wider type just using the equals sign, so

```
double y = 12;
```

so an integer is always promoted to a float or double.

If you want to convert to a narrower type from a wider type you should “cast” the double to an integer, for example.

```
double x = 2.34;
int k = (int)x;
```

If you leave out that cast, the compiler will give you a warning:

```
double y = 12;
float f = y; //compiler warning
float f1 = (float)y; //no warning
```

Arithmetic shortcuts

Instead of writing:

```
k = k + 1;
```

you can compress that, just as in Python, to

```
k += 1;
```

But in C++, you can go farther and just write:

```
k++; // Add 1 to k after you use it
```

And just to be more elaborate, you can also write

```
++k; // Add 1 to k before you use it
```

Of course, you have to be a little careful with this one in writing longer expressions, but it can be very useful.

A complete program

Now, let's look at our first complete program:

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

//Add up the amount of fruit you have
int main()
{
    int apples = 5;
    int oranges = 7;
    int fruits = apples + oranges;

    // print out the sum
    cout << "Total fruit " << fruits << endl;
    return 0;
}
```

Programs in C++ generally begin with a **main** function as we show here. Here are some basic observations:

1. Everything in that main function is enclosed in a pair of braces. Indentation may be more readable but is not required.
2. Single line comments begin with a pair of slashes.
3. The **#include** statement specifies the **iostream** library, much like Python's import statement.
4. The **using namespace std** directive allows you to avoid typing the **std::** prefix before the **cout** and **endl** functions.
5. Output is created using the **cout** function (which stands for "console out." It is commonly pronounced "see out," but many people read it to themselves as "kout.")
6. The main function has a return value of zero if there are no errors. If you return any other number, the operating system will tell you there has been an error.
7. As you might expect, the output of this program is

```
Total fruit 12
```

Bitwise operators

The bitwise operators are intended to do ANDs and ORs and complements on integers to add or mask out individual bits.

<code>&</code>	bitwise And
<code> </code>	bitwise Or
<code>^</code>	bitwise exclusive Or
<code>~</code>	one's complement
<code>>> n</code>	right shift <i>n</i> places
<code><< n</code>	left shift <i>n</i> places

Since bit manipulation may be less familiar to you, here are a few examples. The whole purpose of setting specific bits in a byte or integer is really so you can use that number to set some sort of hardware register or other sort of bitmap.

The bitwise And is sometimes called a masking function. It returns a number that has bits set to one are that are set in both of the input values. So, if we start with

```
int x = 7;           // 0111, and
int z = 10;         // 1010, then
int val = x & z;    // 0010,
                   // since only one bit is set in both
```

The Or operator sets bits in the result that are set in *either* value

```
val = x | z;        // 1111 is the result
```

The complement operator switches all the ones and zeroes in the number.

```
val = ~z;           // 11110101 - to 8 bits
                   // same as -z-1, or -1011
```

The left and right shift operators shift the bits to the left and right, filling with zeroes.

```
val = x << 1;       // left shift 1 place 1110
```

```
val = x >> 1;      // right shift 1 place 0011
```

Types of Integers

The names and lengths of various integer types parallels the history of computer, and microprocessor, development. At one time, integers were 16 bits, but we have pretty much settled on 32 bits (4 bytes) for integers, and 16 bits (2 bytes) for short or short int types. Long or long int may be 4 or 8 bytes. The numeric type list is shown in Table 1-2.

char	1 byte -usually for characters
short	2 byte
int	4 bytes
long	4 or 8 bytes
long long	8 bytes

Table 1-2 – Numeric types in C++

In addition, any of these types may be **unsigned**, meaning that they cannot be negative. This also frees the sign bit of signed integer to allow one more bit of data. This is much less significant today than it was when we were trying to keep memory usage compact in the “olden days.”

Positions of braces

In the example above, you see both the opening and closing braces on separate lines like this:

```
int main()
{
...
}
```

It is also quite common to put the first brace at the end of the previous line. This saves space on the screen or page but is still very readable:

```
int main() {  
    ...  
}
```

We'll use that second convention throughout this book to improve layout on the printed page.

The auto keyword

Since C++ version 11, you have been able to write:

```
double x = 12.3;  
int k = 15;  
auto quot = x/k;
```

The **auto** specifier tells the compiler to deduce the type from the expression. While the result is obvious here, it may not be so clear when you have complex expressions involving pointers to data as we will see in Chapter 7.

This only works when the compiler can deduce the actual type. Otherwise you will get an error message.

Now that we've seen a really simple C++ program, we'll consider some common development environments in the next chapter.

Example code

- `Example1.cpp` -- simple code fragments from this chapter
- `Addfruits.cpp` -- adds apples and blueberries

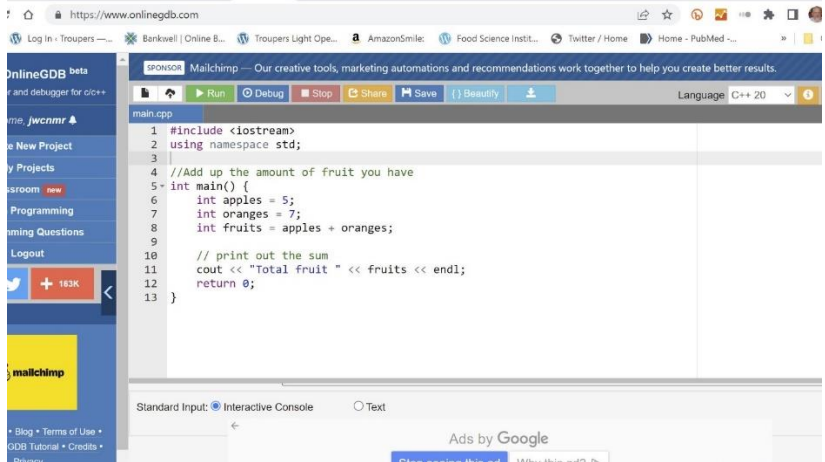
References

If you or one of your colleagues needs a basic tutorial on C++, the free one on the w3schools site is quite good. Look at <https://www.w3schools.com/Cpp/default.asp>

2. C++ Development Systems

If you are going to write C++ programs, you need a code editor and compiler. These are usually bundled together as IDEs (integrated Development Environments). C++ has new version releases about every 2-3 years. The current releases most compilers support are 11, 14, 17, and 20, corresponding to the years 2011 through 2020. The current version is 2023. Here we profile a few IDEs although there are many more.

If you are just starting out, you can't go wrong with a free on-line system like OnlineGDB.



Not only will Onlinegdb work as a C++ development system, it also supports development in C, Java, Python 3, PHP, C#, VB, HTML/Javascript/CSS, Ruby, Perl, Pascal and FORTRAN. In fact it supports versions of C++ from 14 through 20 as well.

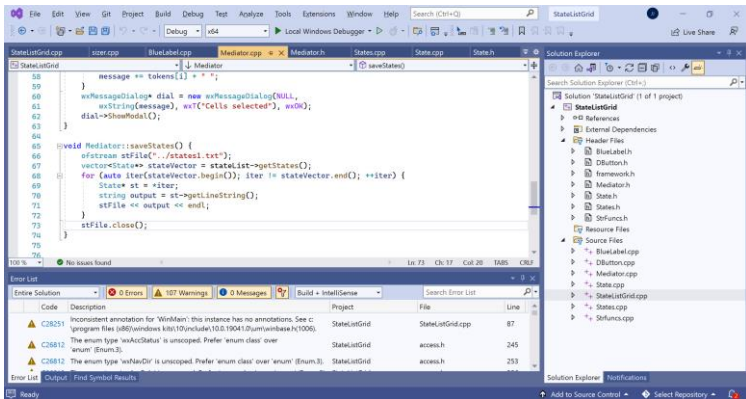
It runs in your browser and compiles and runs code very quickly. It is ideal for beginners, because you don't have to install anything to get started. You do have to create a free account where it can store your programs, And, as you can see, it has a

number of advertisements along the margins to support the project.

You can create programs containing several modules and include files if you read the instructions, but this is not its strength.

Visual Studio

You can download Microsoft's Visual Studio Community Edition from Microsoft for free. And, frankly, it is the easiest to use and most intuitive of all the IDEs we've looked at.

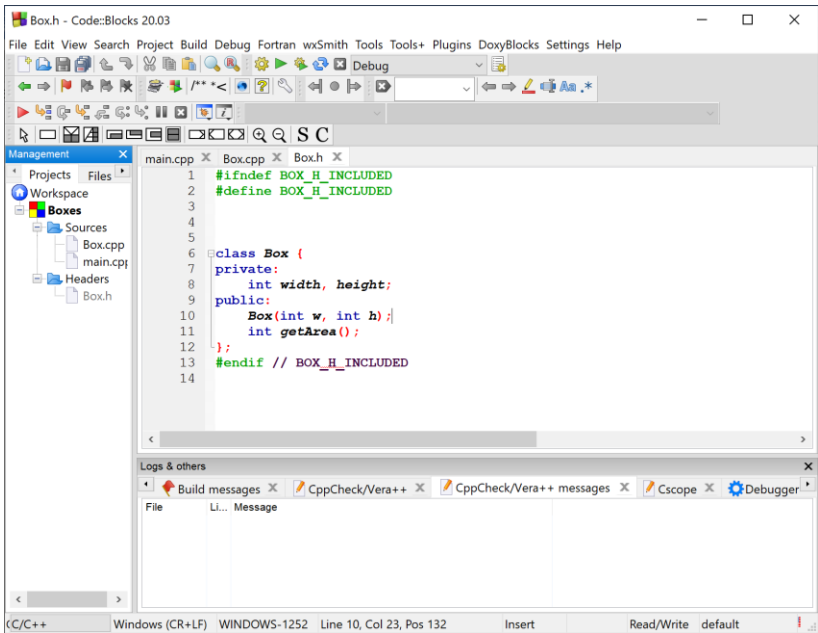


You can use Visual Studio as a development system for C++, C, Python and Visual Basic by simply selecting the type of project you want to build. Creating multi-file projects is pretty easy and you can build and debug programs without much trouble. It is clearly the king of all the IDEs.

CodeBlocks

CodeBlocks is a free download which can either use the gcc compiler or the compiler from Visual Studio. You can build multiple file projects in it and it is recommended in that w3schools tutorial we mentioned. However, we found almost everything we tried to do less than obvious. You need to be careful to select the correct download: the one containing the gcc

debugger is the one you probably want. If you pick one without that debugger, you will get confusing error messages.



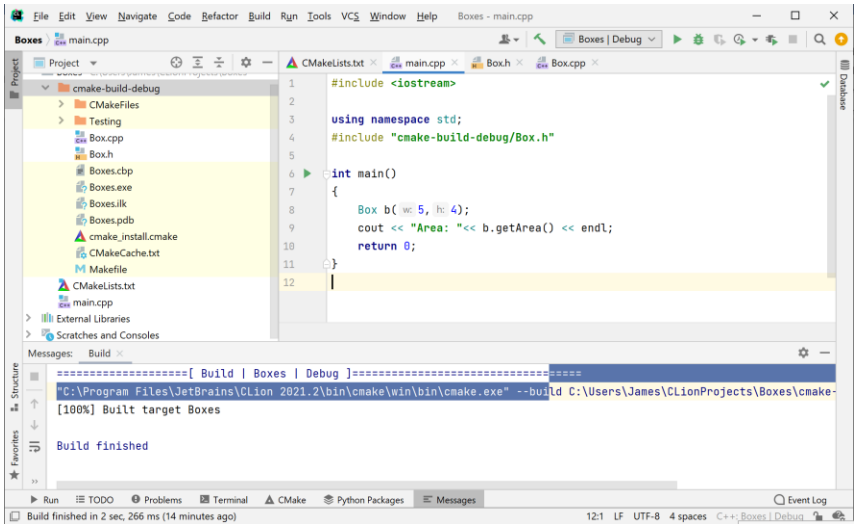
Most IDEs come with several themes which select compatible syntax highlight schemes and backgrounds, but in CodeBlocks, you have to adjust each color yourself. It took quite a long time before we found a setting that would make a decent screen shot. In fact, while the system works quite well, it is not all that intuitive.

CLion

CLion is made by JetBrains, the same company that makes the popular PyCharm IDE for Python. While there is a highly popular free Community Edition for PyCharm, CLion is only free for 30 days, and then it costs \$200 for the first year and a decreasing annual price thereafter.

Nonetheless, it has the advantage of fast startup and compilation that makes it ideal to try out new ideas before integrating them

into larger project on Visual Studio. The IDE is fast and obvious for single file projects and takes only a little time to learn to use for multifile projects.



Other IDE systems

There are at least a dozen IDEs you might consider, reviewed by Dori Esterman in his article “The Best C++ IDEs of 2022.” They include Eclipse, Codellite, NetBeans, DevC++, C++ Builder and Xcode. Any of the above will get you started and you can explore the others at your leisure.

References

1. On Line GDB: <https://www.onlinegdb.com/>
2. Visual Studio: <https://visualstudio.microsoft.com/vs/>
3. CodeBlocks: <https://www.codeblocks.org/>
4. CLion: <https://www.jetbrains.com/clion/>
5. Best IDEs: <https://www.incredibuild.com/blog/best-c-ides>

3. Input and Output

In Python we used the **print** and **input** statements to write and read from the console. These are pretty easy to use unless you need to format the output to a certain number of places. Then you get into formatting. In C++, you use *streams* to do the same thing.. Whether the stream comes from the console or a file, the syntax is exactly the same, making data handling much simpler.

Output using cout

In Chapter 1, our simple illustration used the **cout** object to send a stream of characters to the console:

```
cout << fruit;
```

This syntax uses the less-than signs as a left arrow, meaning to send the string value of the variable **fruit** to the console output. As we noted previously the cout object is supposed to be pronounced C-out (or “see out”) meaning “console out,” but to save syllables, some people just say “kout.”

However, this leaves the cursor at the end of the line. To print a line and move to the next line, using the **endl** object, which is essentially a newline character.

```
cout << fruit << endl;
```

These cout and endl symbols are part of the **std** namespace, which is why our example program begins with:

```
using namespace std;
```

If you decide not to load that huge namespace of symbols into your program, you can also write the statements as:

```
std::cout << fruit << std::endl;
```

Input using cin

Likewise, you can read characters in from the keyboard using the **cin** object followed by two greater-than signs, representing a right arrow:

```
int apples;
cout << "Enter number of apples: ";
cin >> apples;
```

So, our complete program, including the **using** declaration is:

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

//Add up the amount of fruit you have
int main() {
    int apples, oranges;

    //get numbers of apples and oranges from keyboard
    cout << "Enter number of apples :";
    cin >> apples;
    cout << "Enter number of oranges :";
    cin >> oranges;

    //add them together
    int fruits = apples + oranges;

    // print out the sum
    cout << "Total fruit " << fruits << endl;
    return 0;
}
```

Note that unlike Python's **input** statement, the C++ **cin** method converts the input string to the specified simple types *automatically*. You can read into strings, doubles, floats or chars without any special coding.

Reading in whole lines

The **cin** method only reads up to the first whitespace, which might be a space or a Return. If you want to read in a whole line, spaces and all, use the **cin.getline()** method. It reads characters into a **char** array but you can easily convert that to a string.

```
char name[100];           //create a char array
cout << "Enter name: ";
cin.getline(name, 100); //read in a line
string nm(name);        //convert to a string
```

The std namespace

The disadvantage of including the entire std namespace in any C++ program of substantial size is that you might inadvertently create a variable having the same name as one of the many hundreds of keywords in that namespace. But clearly prefixing every input or output method with “std:.” is a significant pain.

There are two solutions to this collision problem. One is to simply create **using** statements for the symbols you need:

```
using std::cout;
using std::cin;
using std::endl;
```

The other solution is to insert the “using namespace std” inside a single function or class, so it is only active in a small, localized code segment.

Formatting in Python

If you write the simple Python code:

```
x = 4.5/3.22
print(x)
```

Python will print out:

```
1.3975155279503104
```

which presents 16 figures, most of which are meaningless. The quotient of these two numbers is irrational, producing an infinite “run-on” decimal result. But such a calculation has at most 2 or 3 significant figures.

But Python doesn't know this. In order to reduce this to just a few significant digits, you can use the popular f-string formatting:

```
print(f'{x:3.3f}')
```

which produces

```
1.398
```

And if you reduce that to two places

```
print(f'{x:3.2f}')
```

you get

```
1.40
```

which, honestly is about the level of precision you should be expecting from 2-digit numbers.

Formatting in C++

The C++ **cout** object handles this a little differently, but there are some similarities.

If you just print out the result of the same operation:

```
double x = 4.5/ 3.22;  
cout << x << endl;
```

C++ prints out:

```
1.39752
```

rather than the long irrational number string you get from Python. Briefly, the **cout** operation has a default width of 6 characters, excluding the decimal point, which is much more friendly.

But if you want to format that number to fewer places, you use the powerful **format** method. For each number you want to format, you create a formatting expression inside a pair of braces. The first character is always a colon. For fixed point numbers like floats, you can specify a width number to the left of a decimal point and the number of digits of precision to the right of the decimal point.

You must include

```
import <format>
using std::format;
```

in your code, and you must be using C++ version 20 or more.

You can have any number of formatted numbers in a single format statement. For each number you want to format, you create a formatting expression inside a pair of braces. The first character is always a colon. For fixed point numbers like floats, you can specify a width number to the left of a decimal point and the number of digits of precision to the right of the decimal point.

Note that the formatting is enclosed in braces, and a list of those variables comes after all those format descriptors and is outside the quotes:

```
cout << format("Both: {:.3f} {:.2f}", x, z);
```

which will, of course, produce:

```
Both: 1.398 0.30
```

```
double x = 4.5/ 3.22;
cout << format("quotient is: {:.2}", x) <<endl;
```

The result is

```
quotient is: 1.4
```

because the format says a total of 2 places. If you want two decimal places, you add an “f” for the float data type:

```
cout << format("quotient is: {:.2f}", x)
<<endl;
```

This produces

```
quotient is: 1.40
```

You can also fold that “endl” into the format statement by using the character symbol for a newline:

```
cout << format("quotient is: {:.2f} \n", x);
```

Symbols in format function

Format has so many options that it is almost a grammar of its own. The main symbols you might use are shown in Table 3-1.

< ^ >	Left, center and right justify
+	Shows the + or – sign
#	An alternate representation for hex and binary
w.	Width if left of decimal point
.p	Precision (floats) right of decimal point
f x b e g	Fixed point, hex, binary, scientific and general formatting
X B E G	Same as x b e g except any letters are capitalized

Table 3-1 -Formatting characters

The argument to the left of the decimal defines the width of the field. So the number is printed with leading spaces in f, e or g format:

```
cout << format("quotient is: {:10.2f} \n", x);
```

results in:

```
quotient is:      1.40
```

The format library picks suitable defaults for each type if you leave out any specifiers. Here we see the default, scientific notation and 12 decimal places illustrated:

```
double weight = 6250.444;
cout << format (
    "weight = {} {:.12e} {:.12f}\n",
    weight, weight, weight);
```

resulting int:

```
weight = 6250.444 6.250444000000e+03 6250.444000000000
```

Hexadecimal (base-16) and binary are convenient ways to print out the bit pattern of variable when they are used as flags. This only works on integers, however. Here we see x and X formats as well as the alternate representation of hex and finally of binary:

```
int y = 9127;
cout << format
    ("y= {:8x} {:8X} {:#8x} {:b} \n",
    y, Y, Y, Y);
```

```
y=  23a7  23A7 0x23a7 10001110100111
```

Alignment and fill formatting

You can align integers and strings inside a wider field using the alignment characters (<, ^, and >), which align the values to the left, center or right. The alignment character must come right after the colon, unless you want to fill the field with something other than a space. In that case the grammar is “: *<” where the field gets filled with asterisks.

Here we see these alignment and fills illustrated for an integer and a string:

```
int k =12;
int j = 20;
string word = "frazzle";
cout << format(
    "k and j= {:3} {:2} {}\n",
    k, j, word);
cout << format(
    "k and j= {:<3} {:*^12} {:>9} \n",
    k, j, word);
```

The result show both the unaligned and aligned values:

```
k and j= 12 20 frazzle
k and j= 12 *****20***** frazzle
```

Error Handling in formatting

The format library is not at all forgiving of errors in the order of the symbols or the types you try to format. If you get the symbols in the wrong order or try to print a string or integer as a float, for example, you will get a runtime error rather than a compile time error. This can be annoying, at least. For this reason, you probably should try the formatting you plan to use on small programs before using it in a larger project.

It is possible to catch the exception that the format object throws and print out an error message:

```
//this one contains an error and will crash:
try {
    cout << "trying to fail \n";
    cout << format("y= {:8.2f}", y);
}
catch (format_error& e) {
    cout << e.what() << endl;
}
```

In the above example, the format function is asked to print out a floating point value, the `y` is an integer. The error that gets printed is pretty helpful:

Precision not allowed for this argument type.

File handling

Files in C++ are just different streams. You can read and write to and from them almost as easily as from the console. If you are reading a file, using the `ifstream` object and if you are writing a file, use the `ofstream` object.

In this first simple program, we read from the file “states.txt” which we use extensively in later chapters. It contains all 50 U.S. states, their abbreviations, capitals and populations.

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

using namespace std;

int main()
{
    const string FILENAME = "states.txt";

    ifstream txtFile(FILENAME);           //open the file
    if (txtFile.is_open()) {              //if it is open
        string line;
        while (getline(txtFile, line)) { //line at a time
            cout << line << endl;        //print out line
        }
        txtFile.close();                   //close the file
    }
}
```

Note that we open the file `states.txt` and read from it a line at a time. This is the first program where we use braces to set off blocks of code inside the larger program. We also indent the code to make it more readable, but that is not required by the C++ compiler, but the braces are.

In this second example we read from one file and write into another, named “mystates.txt.”

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

using namespace std;

int main() {
    const string FILENAME = "states.txt";
    const string OUTFILE = "mystates.txt";

    ifstream txtFile(FILENAME); //open input file
    ofstream outFile(OUTFILE); //open output file

    //if both are open
    if (txtFile.is_open() && outFile.is_open()) {
        string line;
        while (getline(txtFile, line)) { //line at a time
            cout << line << endl; //print each line
            outFile << line << endl; //write to file
        }
        txtFile.close(); //close both files
        outFile.close();
    }
}
```

These examples use the while loop, which is pretty much like the one in Python, and an if statement. We’ll cover them in detail in the next two chapters.

Binary files

Writing and reading binary files is even simpler. You either write them a byte at a time, or an entire dataset at one. The main restriction is that you have to make the compiler believe that you writing an array of bytes, by *casting* the data you are actually writing into a pointer to an array of char.

In this example we create an array of 7 doubles and write them to a file all at once:

```

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

int main() {
    double temps[] = {22.3,35.7,44.8,55.2,61.6,73.8,89.3};

    //write entire file at once
    ofstream outfile;
    outfile.open("temps.dat",
                ios::binary | ios::out);
    outfile.write((const char*)&temps,
                 sizeof(temps));
    outfile.close();
}

```

The resulting file, "temps.dat" is 56 bytes long or 8 x 7.

You can also write the data one data element at a time with a simple for loop:

```

//write a byte at a time
outfile.open("tempsb1.dat", ios::binary | ios::out);
for (int i=0; i< size(temps); i++){
    outfile.write((const char*)&temps[i],
                 sizeof(temps[i]));
}
outfile.close();

```

The file "tempsb1.dat" is exactly the same as the "temps.dat" file.

Reading a binary file

Here we create an empty 7 element double array and cast it in a character array

```

double newtemps[7]; //create empty array
ifstream infile;
// read in data all at once
infile.open("tempsb1.dat", ios::binary | ios::in);

// reads 56 bytes
infile.read((char*)&newtemps, sizeof(newtemps));

```

```
infile.close();
```

Example files on GitHub

- `Addfruits.cpp` – add apples and blueberries
- `Files1.cpp` – reads a text file
- `Filesout.cpp` – writes a text file
- `Fmrs.cpp` – examples of formatting
- `wbin.cpp` – writes and read binary

References

1. <https://www.geeksforgeeks.org/using-namespace-std-considered-bad-practice/>

4. Loops, Arrays and Strings

The C++ array type is analogous to the Python List. But in C++, everything has a type, so C++ arrays must be made up of elements of the same type: integers, doubles, floats and strings. Of course, we will soon see that you can make an array of class objects as well.

In Python, a List is dynamic: you can create it and still append new values of any type later. In C++, however, an array has a fixed size and a fixed type. Once you create it, it remains that size and type. For example:

```
//create an array of integers  
int x[] = {2, 4, 5, 7, 9};
```

Note that you declare `x` as an array by including the empty brackets. In this case, the data are then enclosed in *braces*. After this statement, `x` will always be a 5-member array of integers. You can access array members much as in Python by using indexes in brackets.

```
cout << x[3];
```

This prints out the 4th member, “7,” since all arrays begin with an index of 0, just as in Python.

Of course, you can also create an array and then fill it within your program. Here we create a 10 member array and use a for loop to place numbers in it, starting with 12, and adding 2 to the number each time.

```
int y[10];  
int aval = 12;
```

```
// fill array with numbers starting at 12
// and incrementing by 2
for (int i=0; i<10; i++){
    y[i] = aval;
    aval += 2;
}
```

Then you can print them out just as easily.

```
//print out final array
for (int i=0; i<10; i++) {
    cout << y[i] <<" ";
}
cout <<endl;
```

giving the result:

```
12 14 16 18 20 22 24 26 28 30
```

Two-dimensional arrays

You can represent a 2- dimensional array by just enclosing indices in two successive brackets. Here we create a little 2 dimensional 3 x 4 matrix:

```
double coords[3][4] {
    {3.4, 3.4, 5.5, 2.1},
    {2.2, 1.9, 1.2, 1.0},
    {2.7, 3.4, 4.4, 4.7}
};
```

Then, if you want to access the contents of that array, you refer to each element using the same two brackets. This example prints out the third column:

```
for (int j=0; j<3; j++) {
    cout<< coords[j][2] << " ";
}
```

The result is

```
5.5 1.2 4.4
```

The for loop

You can see from the above examples that you can use the for loop to step through an array. The start and end positions are selectable, and you can adjust the “stride” to any increment you want: you are not limited to 1. And you are not limited to integers:

```
double tval[10];
int i=0;
for(double tmp =42; tmp < 142; tmp +=10.0) {
    tval[i++] = tmp;
}
```

In the above example, we are moving through a set of double precision values starting at 42 and stopping just before the sum reaches 142, with the for loop adding 10.0 each time. The actual array index where we store these values is incremented in the assignment statement just after it is used.

This is all quite different than the Python **for** loop which really only runs sequentially through an iterator.

The range based for loop

There is a version of C++ **for** that does that too, and it is convenient and readable. It’s called the *range based for loop*.

```
for (double a: tval) {
    cout << a <<" ";
}
```

It has the form

```
for (type variable: array_name) {
    operate on variable
}
```

In this approach, you must access the array sequentially: there is no stride option.

The while loops

There are two while loops available in C++;

```
while cond {
    statements
}
```

And

```
do {
    statements
} while cond;
```

Note that the while loop may not be executed at all if the *condition* is false to start with, and the **do while** loop will always be executed at least once.

For example,

```
i = 0;
while (i < size(tval)) {
    cout << tval[i++] << " ";
}
```

may or may not be executed at all and

```
i=0;
do {
    cout << tval[i++] << " ";
} while (i < size(tval));
```

will always be executed once.

Vectors

The vector is much like the Python List. It is a variable length array of any type that you can add to, change or subtract from at any time. The only difference is that in C++, the vector is strongly typed and the type of values must be declared. Inserting any other type of value is an error.

Somewhat awkwardly, the method for adding values to the end of a vector list is named **push_back**.

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

int main() {
    vector<double> vdata; //create a new vector
    vdata.push_back(42.0); //add values to the end
    vdata.push_back(63.2);
    vdata.push_back(77.1);
}
```

Of course, you could also declare the vector contents in a single statement:

```
vector<double> vdat1 {42.0, 63.2, 77.1};
```

You can access any element of a vector just as if it were an array:

```
//print out 2nd and 3rd values
cout <<vdata[1] << " " << vdata[2] << endl;
```

Like Python, you can also pop values off the end of the vector list.

```
vdata.pop_back(); //remove last element
```

But, unlike Python, that value is lost. To get the last value and remove it takes two steps. First save that last value, and then shorten the vector by one:

```
double last = vdata.back(); //get the last value
vdata.pop_back(); //remove it
```

You can, in fact, insert a new value anywhere in a vector. The trick is to get the iterator that points to the first element, and use it to insert. In this example, we insert a new value at the first position.

```
//get the iterator pointing to the first element
vector<double>::iterator it = vdata.begin();
//and insert a new value just before it
it = vdata.insert ( it , 200.4 );
```

Then you can look at the new vector in the usual way using a range-based loop:

```
for (double a: vdata) {
    cout << a <<" ";
}
```

Here is the result. Remember we also removed the last value just above.

```
200.4 42 63.2
```

If you create an empty vector and then add a bunch of values to it, you are asking the vector to reallocate memory every time, and this can be a slow process. It is better to start by creating a vector with a default dimension:

```
//create a new vector and reserve 5 places
vector <double> vdata2(5);
```

This works even if you only use three places. The size of the vector is 3 but the capacity is at least 5. If you create the vector size in advance, you must access the data by its index. Using `push_back` will add data to the end.

Vector Methods

Table 4-1 shows most of the useful vector methods you might use.

begin	Return iterator to first element
end	Return iterator to end
size	Return vector size
max_size	Return maximum size
empty	Return true if the vector is empty
resize	Resize the vector
front	Access first element
back	Access last element
assign	Assign new content replacing current values and size
push_back	Add value to end of vector

pop_back	Remove last value
insert	Insert elements
swap	Swap contents of two vectors
erase	Remove one element of the vector

Table 4-1 - Vector methods

Strings

Strings in C++ are much more flexible than in Python. They are not immutable: you can change characters and add or remove characters much as you can with vectors. In fact, you can access any character using brackets and an index, just as you did with arrays and vectors. The string object is not an integral part of the C++ language, but it was added as a standalone class in C++ 98. Before that time C++ just used C strings which were arrays of char terminated with a null character. You use the string class much as you did in Python.

So, as you have seen, you can create a string

```
string a = "apple"; //create a string
string b("blueberry"); //create another string
```

and access characters like this:

```
//print out the 3rd character of a and the 4th of b
cout<< a[2] << b[3] <<endl;
```

Which prints out

```
pe
```

Note that an individual element of a string is a **char**, not a string of length one.

You can also use the +-sign to combine strings together:

```
string fruits = a + " " + b;
cout << fruits << endl;
```

which, of course prints out the combined string stored in **fruits**.

```
apple blueberry
```

You can also replace characters and insert characters since these strings are not immutable. We change “blueberry” to “blackberry” by changing the ‘u’ and ‘e’ to ‘a’ and ‘k’ and then inserting a ‘c’.

```
//change blueberry to blackberry
fruits[8] = 'a'; fruits[9]='k';
fruits.insert(9,"c");
cout << fruits << endl;
```

Note that the *insert* method takes a string, not a character, so you can insert whole strings anywhere you want to. The string class also has the same **push_back** method you find in the vector class, but note that you can only add single characters with it , not strings, so **insert** is preferable.

Reversing a string

Much is made of reversing a string in Python articles because the articles then teach you the somewhat baroque grammar of string slicing. And since strings in Python are immutable, you always have to create a new reversed string output variable.

In C++ this is much simpler. Since you can switch the characters one at a time, you can do the reversal like this:

```
fruits = a + " " + b;
string revString = fruits; //create the output variable

//and copy one from the front to the other from the back
for(int i = 0, j = fruits.length()-1;
    i < fruits.length(); i++, j--) {
    revString[j] = fruits[i];
}
```

Note that we also illustrate the great power of the for loop. You can initialize more than one starting variable and increment or decrement more than one variable.

You can also use the built-in **reverse** function to reverse a string (or a vector) in place:

```
reverse(fruits.begin(), fruits.end());
```

This works because the string's `begin()` and `end()` methods return iterators rather than counters and the internal code thus has access to the characters of the string itself.

String methods

The string class has a substantial number of methods as we list in Table 4-2. However, a number of the useful ones Python has like `toUpper`, `toLower`, `strip` and `split` are missing. We'll illustrate our own version of these here and later in the text.

begin	Return iterator to beginning
end	Return iterator to end
size	Size of string
length	Length of string (same as size)
max_size	Maximum size of string
resize	Resize string
clear	Clear string
empty	Return true if empty
back	Last character
front	First character
append	Append a string to the string
push_back	Append a character to the string
insert	Insert string into string
replace	Replace portion of string
pop_back	Remove last character
find	Find substring in string
rfind	Find last occurrence in string
find_first_of	Find character in string
find_last_of	Find last occurrence of character
substr	Extract substring
compare	Compare strings

Table 4-2 -String methods

Changing string case

While the string class does not have the familiar Python upper() and lower() functions, the char class does, so you can easily convert a string to uppercase like this:

```
// to upper case
for(int i=0; i< fruits.length(); i++) {
    fruits[i] = toupper(fruits[i]);
}
```

Or, you can use the range-based approach:

```
string word = "BERRY";
int i = 0;
for (char c : word) {
    word[i++] = tolower(c);
}
```

And since C++ is a compiled language, your code will run just about as fast as any built-in method would. We'll illustrate the trim and split methods in Chapter 14.

Converting numbers to strings and vice-versa

Converting from strings to numbers is very easy in C++ using the string-to-integer (stoi), sting to double (stod) and string to float (stof) functions:

```
//convert string to numbers
string snum = "123.4";
double dnum = stod(snum);    //convert to double
float fnum = stof(snum);     //convert to float
int inum = stoi(snum);       //convert to int
```

You can convert any number to a string with the to_string function:

```
//convert number to string
string newst = to_string(dnum);
```

This works for integers of all sizes, doubles and floats. However, doubles and floats are by default represented to 6 decimal places;

```
123.400000
```

even when all those digits are meaningless. If you want to generate a string with fewer decimal places, using the **format** method:

```
string outf = format("{:.2f}", dnum);
```

This will give you two decimal places:

```
123.40
```

[Example Code on GitHub](#)

- **Array1.cpp** – examples of one and 2D arrays
- **Vectordemo.cpp** – examples of vectors
- **Stringdemo.cpp** – examples of strings
- **Stringrev.cpp** – Reverse a string
- **Numconv.cpp** – conversion between string and numbers

5. Making Decisions

The familiar if-else C from Python and Java has its analog in C++. Indentation is common but not required. Many development environments create this indentation for you. However, if there is more than one statement in the if or else blocks, they *must* be surrounded by braces.

```
int apples = 5, berries=7;
int fruit = apples + berries;
if (berries < apples) {
    cout << "add more berries"<<endl;
}
```

If you want to carry out either one set of statements or another depending on a single condition, you should use the **else** clause along with the **if** statement.

```
if (berries < apples) {
    cout << "add more berries"<<endl;
}

else {
    cout << "how about them apples?"<< endl;
}
```

and if the else clause contains multiple statements, they must be surrounded with braces.

Elif is “else if”

When you have a number of choices in a row, such as in the example below, it is helpful to use **if** and then **else if**. The final case can be **else** which covers all the remaining possibilities.

```
int apples = 5, berries=7;
int fruit = apples + berries;
if (berries < apples) {
    cout << "add more berries"<<endl;
}
else if (apples == berries) {
    cout << "still need more berries" << endl;
}
```

```
else {
    cout << "how about them apples?"<< endl;
}
```

Note that in this simple case, you can write this code without braces, since there is only a single statement after each part of the if statement.

```
if (berries < apples)
    cout << "add more berries"<<endl;
else if (apples == berries)
    cout << "still need more berries" << endl;
else
    cout << "how about them apples?"<< endl;
```

Combining Conditions

When you need to combine two or more conditions in a single **if** or other logical statement, you use the logical and, or, and not operators. In C++ you use these symbols rather than the **and** and **or** operators Python expects.

&&	logical And
	logical Or
!	logical Not

In C++ we would write

```
int pears = 2;

if ((apples > berries) || (pears < berries))
    cout <<"enough fruit" <<endl;
```

The Most Common Mistake

Since the “is equal to” operator is “==” and the assignment operator is “=” look very similar, they can easily be misused. If you write

```
//incorrect use of = sign
if (apples = berries)
    cout << "false positive" <<endl;
```

instead of

```
//correct use of = sign
if (apples == berries)
    cout << "same number of each" <<endl;
```

The first case is an error in Python, but legal in C++ because it sets the value of **apples** to the value of **berries**. Since the result is non-zero, this is treated as **true**. This can lead to some confusion in your program, and some IDEs may flag it as a possible error.

Comparing strings

In C++ you can compare strings alphabetically using the same logical symbols. The system compares the strings character by character until there is a difference. If they are the same but one is longer, it is the greater. The comparison is only useful if the strings you are comparing are of the same case, so you usually reduce both strings to lowercase before the comparison:

```
string apple = "apple";
string moreapples = "apples";
if (moreapples > apple)
    cout << "bigger"<<endl;
```

A ticketing program

Let's write a program to determine ticket prices by age. It illustrates how to do a series of if-else if comparisons ending with a final else.

- Ages 6 and under get in free
- Ages 7 to 17 get a student rate
- Age 18 to 59 pay the adult rate
- Ages 60-79 pay the senior rate

- Ages 80 and above get a special, discounted rate

```
int main() {
    // if else demo using ticket prices
    int age = 22; //start with a non zero age.
    while (age > 0) {
        cout << "Enter age :";
        cin >> age;
        int price = 35;
        if (age <= 6)
            price = 0; //toddlers are free
        else if ((age > 6) && (age <18))
            price=15; //student price
        else if ((age >=18) && (age < 60))
            price = 35; //adult
        else if ((age >=60) && (age <80))
            price = 30; //senior
        else
            price = 20; //super senior

        cout << "Price is :"<<price <<endl;
    }
    return 0;
}
```

This little program runs until you enter an age of 0 or less.

The switch statement

The switch statement allows you to do a set of tests on integers or characters and execute code based on the value. It doesn't allow ranges or strings, however, but it can be very useful when you want to distinguish small whole numbers. It is a simpler ancestor of Python's match statement, but can be helpful, for example, in distinguishing one character commands.

It has the form

```
switch (num) {
    case 1:
        statement1;
        break;
    case 2:
```

```

    case 3:
        statement2;
        break;
    default:
        statementdeflt;
}

```

There can be one or more statements in each case since each case is terminated with a break statement. You can also have several cases that call the same statement as we show for statement2.

For a working example, let's redo our ticketing program using single character commands for each age group:

- C for child
- Y for youth
- A for adult
- S for senior
- 8 or o for 80 or over

Here's the entire program:

```

int main() {
    char c = 'a';
    //get the first character of whatever is entered
    while (c != 'q') {
        cout << "Enter age group: (c, y, a, s, o) : ";
        string ageGroup= "a";
        int price = 0;

        cin >> ageGroup;

        if (ageGroup.length()>=0) {
            //convert to lowercase
            c = tolower(ageGroup [0]);
        }
        switch (c) {
            case 'c': //child is free
                price=0;
                break;
            case 'a': //adult
                price = 35;
                break;
            case 'y': //youth
                price=15;

```



```

        break;
    case 's':          //senior
        price = 30;
        break;
    case 'o':          //octogenarian
    case '8':
        price = 20;
        break;
    default:          //everyone else
        price = 35;
    }
    cout << "Price=" << price << endl;
}

```

Break and continue

Like Python and Java, C++ has the break and continue statements that allow you to jump out of the middle of loops. If you agree with the principles of structured programming, then a loop should have only one entrance and one exit point, the break and continue are confusing additions to the language.

- Break jumps completely out of the loop to the next statement, and
- Continue jumps to the bottom of the loop no matter what

Let's look at a couple of examples. This program exits the summation loop as soon as the sum exceeds 15, producing a sum of 19.

```

double xarray[] = {5, 7, 4, 3, 9, 12, 6};
double sum = 0;
for (double x: xarray) {
    sum += x;
    if (sum > 16) {
        break;
    }
}
cout << "Sum1 = " << sum << endl;

```

This code does the same thing, but uses a **quit** flag to decide the exit form the while loop:

```
//alternate approaches
bool quit = false;
sum =0;
int i = 0;
while (! quit) {
    sum += xarray[i++];
    quit = sum > 16;
}
cout << "Sum2 = " << sum <<endl;
```

This is much cleaner and more readable. And to simplify this further, you could have written this more directly as:

```
sum =0; i=0;
while (sum <=16) {
    sum += xarray[i++];
}
cout << "Sum3 = " << sum <<endl;
```

All three examples give the same result, and the last two are surely clearer.

We might use the **continue** statement to skip an element in an array:

```
//continue statement
for (i=0; i< size(xarray); i++) {
    if (i ==4) continue; //skips index 3
    cout << xarray[i]<<" ";
}
cout <<endl;
```

This one skips element 4, the 5th member, the number 9. The resulting printout is

```
5 7 4 3 12 6
```

A cleaner way to write this is to just skip the value 9:

```
for (double x: xarray) {
    if (x !=9) cout << x <<" ";
}
```

This gives the same answer and is much clearer.

The ornery ternary operator

C++ and Java (but not Python) also feature the ternary operator which allows you to make decisions in a single statement. It is mostly of historical interest, since it is pretty hard to read and decipher. It has the form:

```
variable = (expression) ? var1 : var2;
```

If the expression is true, the variable is assigned the value of var1 and if false it is assigned the value of var2. It is exactly the same as

```
if (expression) {
    variable = var1;
}
else {
    variable = var2;
}
```

Here's an example:

```
int main() {
    int berries =100;
    int beans = 50;

    int produce = (berries > beans) ? berries : beans;
    cout <<"produce is: " << produce << endl;
    return 0;
}
```

Since this is exactly the same as the more readable:

```
if (berries > beans)
    produce = berries;
```

```
else
```

```
    produce = beans;
```

```
cout <<"produce is: "<< produce << endl;
```

we never use it. This statement is of historical interest, when compilers were less sophisticated, but today compilers produce the exact same code for both code snippets. The ternary operator is great fun in producing “obfuscated C code,” however.

Example programs on GitHub

- Decisions.cpp – examples of if-else if-else code
- Ticketages.cpp – example of ticket age code
- Switcher.cpp – switch examples
- Breakcont.cpp – illustrates break and continue
- Ternary.cpp – ternary operator example

6. Functions

Functions are a significant part of C++ and most other languages. They are units of code that carry out a specific set of operations. And while functions can be called many times throughout a program, there are plenty of cases where a function is called just once, but conveniently groups a set of operations that you need to call while starting up a program.

Functions are usually called with one or more arguments and may return some value when they exit. To declare a function, start with the name of the function followed by parentheses. The entire function is enclosed, of course, in braces. Most development systems indent the code inside the function much as they do the contents of a loop. Let's write a really simple function first, that calculates the square of a number:

```
//square the argument and return it
double sq(double x) {
    double y = x * x;    //create the square
    return y;           //and return it
}
```

Note that functions that return a value have their type declared first thing, followed by the function name and then the parentheses, which may contain one or more arguments, each with their type. Functions begin at the opening brace and end at the closing brace. If they return a value, they use a **return** statement.

The variable `y` inside the function is a *local variable*. It has no existence outside the function's braces. And in fact, in this simple case, you could just omit it:

```
//square the argument and return it
double sq(double x) {
    return x * x;        //and return it
}
```

Of course, functions can call other functions. We could create a **cubed** function that calls the **sq** function:

```
//cube the argument and return it
double cube(double x) {
    double y = sq(x);
    y = y * x;    //create the cube
    return y;    //and return it
}
```

Then we call those functions from our main program:

```
int main() {
    double asqd = sq(12.0);
    double cubed = cube(12.0);
    cout << asqd <<" " << cubed << endl;
}
```

Function order

But C++ has a particular rule about the order of such functions:

1. Generally, the **main** function must come last.
2. If one function calls another function, that second function must already have been declared in your code.

In other words the C++ compiler builds a symbol table in the same pass as the compilation, and it must have already encountered any additional functions. So, in this case, the functions must appear in the order:

- **sq()**
- **cube()**
- **main()**

So **cube** can call **sq** but not the other way around, and **main** can call both. We'll see how you work around this issue shortly.

Overall, our little program looks like this:

```
using namespace std;

//square the argument and return it
double sq(double x) {
    double y = x * x;    //create the square
    return y;    //and return it
}

//cube the argument and return it
double cube(double x) {
```

```

    double y = sq(x);
    y = y * x;      //create the sube
    return y;      //and return it
}

int main() {
    double asqd = sq(12.0);
    double cubed = cube(12.0);
    cout << asqd <<" " << cubed << endl;
    return 0;
}

```

Polymorphism in functions

You can have more than one function with the same name as long as the arguments are different in type or number. So you could also write:

```

int sq(int x){
    return x*x;
}

```

and that function will not collide with the double version, since the arguments are different. In C++, unlike Python, the function signature include the types and number of arguments, and they can co-exist as long as the signatures differ.

Function prototypes

You could create a **power** function, and then have the square and cube call it:

```

double power (double x, int pwr) {
    double y=1;
    for (int i=1; i<= pwr; i++){
        y =y * x;
    }
    return y;
}

```

But eventually you are going to be in the position where several functions call each other, and there is no obvious order to put them in so each knows about the others.

In this case, we resort to *function prototypes*. We simply declare all of the function names and types at the beginning of the program, but without the function body:

```
double sq(double x);
double cube(double x);
int sq(int x);
double power (double x, int pwr);
```

Then you can place the functions anywhere you want, usually after the **main** function. This way every function, including **main** knows about every function in the program.

Passing arguments to functions

If we create the function `sq1`, which operates on its argument:

```
double sq1(double x) {
    x = x*x;
    return x;
}
```

we appear to be changing the argument itself. But if we look at the result of this simple call:

```
double x =12;
double y = sq1(x);
cout << "y="<<y<<" x="<<x<<endl;
```

The resulting output is:

```
y=144 x=12
```

In other words, even though we changed `x` inside our **sq1** function, `x` in the calling program is not changed. This is true in Python as well, and in both cases it is because `x` is *copied* into the function. So the value of `x` is passed in, not the original `x` variable in the calling program. This is referred to as *call by value* because the variable's value is passed in, not the variable itself.

In Python, simple variables are always passed by value, and larger mutable objects are passed by reference.

This is not true in C++. All variables other than arrays are copied into the function, and arrays are actually pointers as we'll see next.

Default arguments

You can also create functions with default arguments. For example:

```
void errmsg(string text ="error in program") {
    cout <<text<<endl;
}
```

If you call this function with no arguments, it uses the default value of `text` shown in the function declaration.

```
errmsg(); //prints out "error in program"
```

But if you call this function with a new message

```
errmsg("both arguments are zero");
```

it prints that out instead,

You can do this with numeric arguments as well:

```
double area(double x, double y =0){
    double retval = 0;

    if (y !=0) retval= x*y;
    else retval= x*x;

    return retval;
}
```

Then a single argument returns a square and two arguments returns their product:

```
cout << area(12,14) << endl; //product
cout << area(12) << endl;    //square
```

Using constant declarations

You already know that you can declare a value as being constant using the **const** declaration:

```
const ultimate = 42;
```

But you can use this same declaration to indicate that some function arguments cannot be changed. Suppose you wanted to pass an array to a function and have it compute the mean value. If you declare that the array is a constant, your function cannot change any members of the array:

```
double meanValue(const double x[], size_t size) {
    double sum = 0;
    for (size_t i=0; i <size; i++) {
        sum += x[i];
    }
    //x[0] = -1; //read only variable--error
    return sum / size; //calculate average
}
```

In fact, if you try to set an array value as we show in the comment, the compiler will flag this as an error because the array is now *read only*. You can do the same kind of things with class members and entire classes as we will see shortly.

It is relatively uncommon for programmers to write functions which change their arguments. Instead, functions return values. For this reason, the **const** declaration is frequently omitted. But if you are writing code others may use, it is wise to include them.

Example programs

- `functs.cpp` – illustrates sq and cube functions
- `funcproto.cpp` – shows how to use function prototypes
- `funcdefault.cpp` – shows default arguments
- `constExample.cpp` – the mean value example using `const`

7. Using Pointers

Pointers appear promiscuously in the C++ and C languages and are variables containing the address of some other variable or structure. They are particularly useful when you want to pass a large object without the overhead of copying it.

While the idea of pointers seems daunting to some, we can cover it in three lines. Suppose you create a string object

```
string mtb = "meatball";
```

and want to get its address. We use the reference operator or ampersand:

```
string* pmeat = &mtb; //pointer to string
```

The new variable **pmeat** is a reference or pointer to the mtb variable and has the type “string*” which means *pointer to string*.

Now, if we want to get the value of that variable, we use this same *-operator to *dereference* the pointer, getting back the original contents:

```
cout << *pmeat <<endl; //print out the string text
```

Here we see the “*pmeat” means get the value the pointer points to. And if run this C++ will indeed print out

```
meatball
```

as you expect.

To summarize:

- ‘&’ gets a pointer to a variable.
- ‘*’ gets the value pointed to by the pointer (or dereferences it.)

The only point of confusion is that C++ also uses the “*” symbol in declaring a variable’s type.

```
double* pval;
```

means that `pval` is a pointer to a double value. You can also write this as

```
double *pval;
```

which has exactly the same meaning.

Arrays and pointers

If you create an array of doubles, like this

```
double xarray[] = {12, 14, 15, 16, 20};
```

the array name `xarray` is really a pointer to the memory the begins the storage of the array. Here there are 5 8-byte double precision numbers stored in 40 consecutive bytes.

So you could copy that pointer, and then increment it to point to each of the 5 elements:

```
double* px = xarray;
for (int i=0; i<5; i++) {
    cout << *px++ <<" ";    //array pointer incremented
}
```

In the above example, `px` is a pointer to the beginning of the array, just as `xarray` is. But if we increment it, it will point to each successive data element in the array. And the size of that increment is determined by the data type. So `*px` points to the current array element, and if we increment after we use it, it will then point to the next number.

The loop prints out the result:

```
12 14 15 16 20
```

by moving the pointer through the array.

Calling functions

We have already seen that a function like this one

```
//call by value.
// Changes only within the function
void getReal(double x) {
    x=15;        //changes only inside function
}
```

does nothing to the calling parameter. If you call the function

```
double y {172.6};
getReal(y);        //pass by value
```

the variable `y` is unchanged. This is true of all single-value variables.

But, if you pass a pointer to `y` into this function:

```
void getPreal(double* px) {
    *px = 22;        //changes the calling parameter
}
```

and call the function with a reference to `y`,

```
getPreal(&y);        //pass by reference
```

the value of `y` in the calling program *is* changed, because you dereference the pointer with that “*” so you are changing the original calling program’s variable. In other words, **p points to* the variable in the calling program, and you can change it from within the function. This is called *call by reference*.

In this simple case, this just looks malicious, but in real programs that variable might be an instance of a class, and it is not unreasonable that a function might want to change some value inside a class. So, this is much more useful than it first might seem.

Functions and Arrays

If you pass an array into a function, you are passing in the pointer to the beginning of the array. Using the same **xarray** we created above, we might want to call this function:

```
changeArray(xarray);
```

And here is the function

```
void changeArray(double *xa){
    xa[4] = 42.0;    //change one value
}
```

The **xarray** variable points to the beginning of the array, so we can use the above pointer to form the array expression. In fact, we could also add 4 to the pointer to get the same result:

```
*(xa+4) = 42.0;
```

You could also create a function that has the array itself as an argument:

```
void changeAnArray(double xarray[]){
    xarray[3]=666;
}
```

But it is actually just another way of writing the same thing.

One problem with passing arrays into functions, is that only the pointer to the start of the array is passed in. There is no information on the array's actual size, and while the compiler will let you write:

```
void changeAnArray(double xarray[]){
    xarray[300]=666;
}
```

this will fail at run time because that is probably outside the bounds of the array. One way to get around this is to pass the array size into the function as well:

```
void changeArray(double *xa, size_t size){
    xa[4] = 42.0;
}
```

Of course, you must then check the index you use against that size:

```
if (index < size) {
    xa[index] = 42.0;
}
```

But, you can get around all of these restrictions by using **vectors** instead of arrays. Vectors carry their size information (which may expand as needed) along with them.

If you create a vector and pass it to a function that changes values in it:

```
vector<int> v = { 3,6,7,8,12 };
changeVec(v);
```

and that program changes the value of one element:

```
void changeVec(vector<int>px) {
    if (2 < px.size()) {
        px[2] = 123;
    }
}
```

the resulting vector is copied into the **changevec** function and only that copy is changed. The vector in the calling program is unchanged.

Obviously copying large vectors around isn't ideal, but if you send a pointer to the vector into the function, it changes the original vector:

```
void changeVec(vector<int>& px) {
    if (2 < px.size()) {
        px[2] = 123;
    }
}
```

C strings and pointers

In the C language, strings were represented as an array of characters, terminated by a zero or null character. For example:

```
char greeting[6] ={"hello"}; //C string is an array
```

The actual array size must be one greater than the number of characters to make room for the '\0' terminating character. And just as with the numeric arrays we've been dealing with, the address of that string is a pointer to that array. So you could print out the string a character at a time using a pointer:

```
char* p1 = greeting;
for (int i=0; i<5; i++) {
    cout << *p1++;
}
cout <<endl;
```

Now, by contrast, strings in C++ are actual classes and you can print them out without much thought. However, if you come across some old function that requires a C string, you can get one using the `c_str()` method:

```
//a C++ string
string cpstring("This is a C++ string");
cout <<cpstring <<endl;

//get the C string within
const char* cpp = cpstring.c_str();

for (size_t i=0; i< strlen(cpp); i++) {
    cout << cpp[i];
}
cout <<endl;
```

However, the C-string you get is constant or immutable. To get that same C-string so you can alter it, you can use the `data()` method:


```
//get a mutable version of that C-string
char* vcpp = cpstring.data();

vcpp[2] = 'u';           //change one character
cout <<vcpp <<endl;
```

Remember that C-strings are mainly of historical interest, and you will seldom use them. The preferred C++ string is the **string** class.

[Example code on GitHub](#)

- `Pointers.cpp` – contains all the code examples in this chapter
- `ChangeVec.cpp` – changes vector inside function
- `Charpointer.cpp` – illustrates C string

8. Sets, tuples and maps

Sets

Sets in C++ are very much like those in Python. The difference is that the members must all be of the same declared type. The main use of a set is to create collections of items which have no duplicate members. If you try to add another item to a set that already holds that item, it will not be added.

There are actually two set objects in C++: **set** and **unordered_set**. The usual set is always stored in ascending order, while the **unordered_set** which is backed by a hash table for controlling duplicates. For small sets, it doesn't make much difference which you use. For large sets, the **unordered_set** may run faster. Be sure to run a timing test on your data to be sure.

Creating sets is incredibly simple.

```
set<int> cset={2,5,12}; //create a set
cset.insert(5);      //add a duplicate
cset.insert(6);      //add a new number
```

Now, if we print out that set:

```
//print out set contents
for(int s: cset){
    cout << s << " ";
}
cout << endl;
```

We will find that the set only contains one 5.

```
2 5 6 12
```

If we want to find out if the set contains a particular value, the **find** method will do it for you.

```

//check to see if set contains a 6
int num = 6;
auto it = cset.find(num);
if (it != cset.end()){
    cout << num << " is in the set"<<endl;
}
else
    cout << num << "is NOT in the set" <<endl;

```

The **find** method returns an iterator which points to the position of that value in the set. But if that value is not found in the set, the iterator points to the end of the set. Hence by comparing with the **end** iterator, you determine whether the value was found or not.

Sets are not limited to integers, of course. They can be made up of strings:

```
set<string> stSet{"Fred", "Nora", "Zoltan"};
```

or doubles:

```
set <double> dSet {22.4, 6.02e23,1.008};
```

Merging sets

Merging sets is not too difficult, although it should be easier. You create the sets and then insert all of the second set at the end of the first set.

```

set <string> fruits{"apples", "pears", "cherries"};
set <string> piestuff{"nuts", "berries", "apples"};
set <string> pie = fruits ;
pie.insert(piestuff.begin(), piestuff.end());

```

Merging these two creates a set with 5 members, since apples occurred twice:

```
apples berries cherries nuts pears
```

There are articles on how to compute the intersection of two sets that you can find on line, but they are too complex to take up here.

Tuples

Tuples in C++ are much like those in Python, a collection of values of different types that cannot be changed or added to (immutable). They are a convenient way to return more than one value from a function and you can create them rather simply.

First, you can declare a tuple and all its types like this:

```
tuple <int, string> breadTuple(12, "loaves");
```

and secondly, you can use the **make_tuple** function to create the tuple, and deduce the types in the process:

```
auto newTuple =
    std::make_tuple ("Sarah", "Snoodly", 14, 'y');
```

Note that we've used the **auto** type to tell the compiler to create the needed type without you spelling it out. But the main thing about the statement shows how simple a tuple really is to create.

However, fetching values from a tuple is unnecessarily complicated. You cannot use a variable to specify the index into the tuples, here numbered 0 to 3. Instead, you must fetch them using the constants 0 to 3:

```
// You must access the tuples with a constant
index, not a variable
cout<< get<0>(newTuple) <<" ";
cout<< get<1>(newTuple) <<" ";
cout<< get<2>(newTuple) <<" ";
cout<< get<3>(newTuple) <<" ";

cout << endl;
```

Fortunately, there is a convenient workaround. You create a set of variables representing the values within a tuple:

```
string fname, lname;
int age;
char honors;
```

Then you can use the `tie` function to copy the contents of the tuple into those variables:

```
//copy the tuple members into the variables
// to make them easier to print out or use.
std::tie(fname, lname, age, honors) =
    newTuple;
```

Or, starting in C++ 17, you can copy them into new variables created on the spot as you see here:

```
auto [fname1, lname1, age1, honors1] =
    newTuple;
```

In this case, you don't declare the variables in advance: they are created with the proper types because we used `auto` to say that we want the compiler to deduce the types of those variables.

One of the most controversial parts of discussing tuples is how to pronounce them. Now Easy Reader would probably suggest too-pul, since the first syllable ends in a vowel, and in many cases in English that means the vowel should be long. But computer geeks have decided it should be pronounced tuh-pul, as if there were two p's instead of one. The rationale for this is that triple, sextuple and septuple are pronounced with a short 'u' so it should be, too. This of course ignores quadruple and octuple which usually have a long vowel pronunciation. But you can pronounce it any way you want. If in spoken discourse, someone corrects you, just remember that if the British can pronounce the surname Cholmondeley as "Chumley," all bets are off.

Maps and Dictionaries

C++ does not have a dictionary type like Python does, but the **map** type is very similar. While Python dictionaries are usually made up of a string key and a string value, you have more flexibility in C++, since you must declare the types of the key and value in advance:

```
map<string, string> dbanswer;
```

Then you can insert values into the dictionary like this:

```
dbanswer.insert(pair<string, string>("fname",
    "Sally"));
dbanswer.insert(pair<string, string>("lname",
    "Splurge"));
dbanswer.insert(pair<string, string>("score",
    "98"));
```

and if you want to fetch a value using the key, you simply fetch it using the key:

```
cout << dbanswer["score"] << endl;
```

If you aren't sure that the map contains a value with that key, you can check it using **find**. If **find** returns an iterator pointing to the end, the key is not found.

```
string keyScore = "score";
auto it = dbanswer.find(keyScore);
if (it != dbanswer.end()){
    cout << dbanswer[keyScore] << endl;
}
```

You can also print out the entire dictionary entry in a simple loop like this:

```
string keys[] = {"fname", "lname", "score"};
for(int i=0; i<3; i++){
```

```
    cout <<keys[i] << ": " << dbanswer[keys[i]] << endl;
}
```

A more compact way of adding a list of pairs to a map is shown below.

```
map<string,string> states;
states["AR"] = "Arkansas";
states["AK"] = "Alaska";
states["CA"] = "California";
states["CT"] = "Connecticut";
states["MO"] = "Missouri";
states["KS"] = "Kansas";

cout << "CT: "<<states["CT"]<<endl;
```

It would be possible to create a map within a map where the inner map contains other state properties, like capital and population, and then create an outer map of these property maps, but it is probably better to create little State classes instead. We'll take up classes in the next chapter.

Example programs in GitHub

- **setsTuples.cpp** – examples of sets and tuples
- **maps.cpp** – examples of using maps
- **setandTuple.cpp** – shows creation of tuples.

9. Classes and OOP

Classes in C++ are similar to the ones you learned in Python, but they are more flexible than Python's are. Quite a few basic C++ books and tutorials put off covering classes until so late in the book that some newcomers have gotten the idea that classes are somehow optional add-ons, and they put off learning them at all.

But they have gone at this wrong-way round. Almost every component of C++ is an object.

- Objects hold data and have methods to access and change that data

For example, strings, tuples, sets and maps are all objects, as are complex numbers. And they all have functions associated with them called *methods* that allow you to get and change that data.

The whole idea of data inside classes is called *data encapsulation*. You don't need to know how the data are represented or computed: you just use the getter and setter methods to obtain and store that data.

But how do you make your own objects?

A Rectangle class.

You create objects by first defining a class which describes that object. Let's start with a simple example that draws a rectangle. To create a class you use the **class** keyword followed by a name.

```
class Rectangle {
    int width, height;
public:
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }
};
```

Much like the similar Python class, this Rectangle class begins with a **class** declaration followed by the class name. It is neither customary nor frowned upon to capitalize class names. It does help set them off in the code, though. The code in the class begins after a left brace and ends with a right brace *followed by a semicolon*.

The class instance variables are usually declared first and are by default **private** whether you declare them as private or not. This means that they can only be accessed by code within the class. If you want to derive new classes from this class and want them to be able to access these variables, you declare them as **protected** instead. In C++ it is conventional to keep all these variables private or protected and allow programs to obtain those values with **get** methods like this one:

```
int getWidth() {
    return width;
}
```

In C++, the constructor copies the variable values into the class instance, much like the `__init__` method in Python. The constructor has the same name as the class but copies the arguments into the variables `width` and `height`.

```
public:
    Rectangle(int w, int h): {
        width = w;
        height = h;
    }
```

You can also use the slightly shorter brace notation to indicate copying these variable values:

```
public:
    Rectangle(int w, int h): width{w}, height{h} {
    }
```

Our Rectangle class can print out a rectangle made up of asterisks and spaces. We add the spaces in the top and bottom lines to match the spacing between lines. We could print the top (and bottom) lines of the rectangle by printing out an asterisk followed by 2 spaces for each element of the width:

```
void drawTop() {
    for (int i=0; i<width; i++) {
        cout << "*"  ";
    }
    cout <<endl;
}
```

But it might better to make that asterisk and spaces string into a constant, along with 3 pure spaces we'll need for the middle lines:

```
const string star="*  ";
const string spaces ="   ";
```

Then our **drawTop** method beomes:

```
void drawTop() {
    for (int i= 0; i < width; i++) {
        cout << star;
    }
    cout << endl;
}
```

and we draw the sides with a similar method:

```
void drawSides() {
    for (int i=0; i<height-1; i++) {
        cout << star;           //left side
        for (int j=0; j<width-2; j++){
            cout << spaces;
        }
        cout << star << endl; //right side
    }
}
```

and we can draw the whole rectangle with this simple method:

```
//draws whole box
void draw() {
    drawTop();    //top
    drawSides(); //sides
    drawBottom(); //bottom
}
```

And here is the entire 5 x 8 rectangle:

```
* * * * *
*           *
*           *
*           *
*           *
* * * * *
```

Inheritance

Drawing a square is, of course, a special case of Rectangle. We can derive a Square class from the Rectangle class by simply passing the side dimension into the constructor twice:

```
//square derived from Rectangle
class Square:public Rectangle{
public:
    Square(int size):Rectangle(size, size){}
};
```

And that's it. That's the whole class. The **size** argument is passed to the Rectangle constructor. Everything else is the same! Here's the proof:

```
* * * * *
*           *
*           *
*           *
*           *
* * * * *
```

More useful classes

Rather than deal with cute Dog or Car classes, let's instead get to work with a useful class describing an employee. Our employee class will contain the employee's name, salary, benefits, and an ID number. Here's the beginning of the C++ class for an Employee:

```
class Employee {
private:
    //private variables
    int idNum;
    string fname, lname;
    double salary;
    double benefits;
public:
    //constructor initializes variables
    Employee(int id, string frnm, string lnm,
             double sal, double ben = 1000){
        fname = frnm;
        lname = lnm;
        idNum = id;
        salary = sal;
        benefits = ben;
    }
    //return salary
    double getSalary() {
        return salary;
    }
};
```

The **public** section of the class contains all the methods that other classes can access and, most important, contains the constructor. Much like the Python `__init__` method, the constructor sets the values of many or most of the class's private variables as you see above. There is a second style of syntax for the constructor that is a little more compact:

```
public:
    Employee(int id, string frnm, string lnm, double sal):
        idNum{id}, fname{frnm}, lname {lnm}, salary{sal}
    {
    }
```

Note that these values are assigned to the class variables after a colon and before the open left brace of the (now empty) body of the constructor.

You can even use this brace construction to initialize the class variables so that have actual values ahead of time:

```
int idNum{0};
string fname{NULL}, lname{NULL};
double salary{0.0};
double benefits{1000};
```

These are called *braced initializers* and are in most cases the same as using equals signs. The one difference is in the case when you inadvertently write an initializer statement that narrows the value. For example:

```
int x = 4.5;    //always legal in C++
int x {4.5};   //compiler will issue a warning
```

If you convert a double or float to an integer, this might be a mistake, and the compiler will issue a warning if you use the braced initializer.

Of course, you can have default values in the constructor as well as in any class methods. For example, the **benefits** value might have a default value like this:

```
Employee(int id, string frnm, string lnm,
          double sal, double benefits = 1000) {
```

but this presents some style problems. We used abbreviated names for the other variables, but for one whose name might be spelled out in the calling program

```
Employee emp1 = Employee(id++, "Susan", "Sugar",
                          5000, benefits=1000);
```

we want to use that full name. Then, what do we do with the class member variable names? One solution is to prefix the internal member names with an `m_` as we do here:

```
class Employee2 {
private:
    int idNum{0};
    string m_fname{NULL}, m_lname{NULL};
    double m_salary{0.0};
    double m_benefits{1000};
public:
    Employee2(int id, string fname, string lname,
              double salary, double benefits=1000) :
        idNum{id}, m_fname{fname}, m_lname{lname},
        m_salary{salary}, m_benefits{benefits} {

        }

    //return the current salary
    double getSalary() {
        return m_salary;
    }

    //return the name
    string getName() {
        return m_fname + " " + m_lname;
    }
};
```

This makes using those names internally a little more awkward but makes the names of the variables in the constructor more obvious.

Another approach is to use the “hidden” **this** pointer which points to that class instance. It is pretty much like the **self** variable in Python except that it is a pointer:

```
Employee(int id, string frnm, string lnm, double sal,
         double benefits = 1000) {
    fname = frnm;
    lname = lnm;
    idNum = id;
    salary = sal;
    this->benefits = benefits;
}
```

You could do this for all the variable names or just for the default ones, as you prefer. But you must make the names in the constructor argument and the names of the internal variable somehow different.

Deriving new classes

Now let's consider some other types of employees. For example, we might have temporary employees who get reasonable salaries, but no benefits. We can derive a new TempEmployee class in just a few lines:

```
class TempEmployee : public Employee {
public:
    TempEmployee(int id, string frnm, string lnm,
                 double sal) :
        Employee(id, frnm, lnm, sal) {
        benefits = 0; //temp employees no benefits
    }
};
```

That's the whole class. The only difference is that the constructor sets the benefits value to zero.

We might also have an Intern class, who not only gets no benefits, but has a salary cap, since these are essentially trainees.

We do this by creating a little private **capSalary** method that sets any salary proposed above 500 to 500.

```
void capSalary() {
    if (salary > 500) {
        salary = 500;
    }
}
```

So, our entire Intern class, which is also derived from the Employee class makes two changes: benefits are zeroed out and the salary is capped:


```

class Intern : public Employee {
private:
    //cap the salary at 500 no matter what was entered
    void capSalary() {
        if (salary > 500) {
            salary = 500;
        }
    }

public:
    Intern(int id, string frnm, string lnm, double sal) :
        Employee(id, frnm, lnm, sal) {
        benefits = 0;           // no benefits either
        capSalary();           //cap the salary
    }
};

```

Public, protected and private inheritance

In the above examples (and most examples) we show the base class name preceded by the **public** keyword.

```
class Intern: public Employee {
```

However, you can if you wish, use the **protected** or **private** keywords to define the inheritance instead.

- In public inheritance, the base class public variables are public in the derived class and protected variables are protected in the derived class.
- In protected inheritance, base class public and protected variables become protected in the derived class.
- In private inheritance, base class public and protected variables are private in the derived class.

Note, however, that if you leave out that **public** modifier, it defaults to **private** and your derived classes will not have access to any of the base class variables.

Classes within a class

Now suppose we have a small group of employees and want to display them or do some calculations on the group. It would be nice we could create a class that represents all the employees. This is just a simple as it seems: we just add each new employee to an array, a vector or some other container, so we can run through them quickly.

You might think that a vector of Employee objects would be just the thing, but that stores copies of the Employee objects rather than the originals. How do we solve this? By using pointers some more. In fact, this is the most common use of pointers in C++.

When we create an Employee object like this:

```
Employee emp = Employee(id++, "Susan", "Sugar", 5000);
```

the compiler reserves memory for that object at compile time, and that memory is relinquished when the program exits.

But if you wanted to create pointers to a bunch of Employee objects you could use the **new** operator which reserves that memory at run time:

```
Employee* emp1 = new Employee(id++, "Susan", "Sugar",  
                               5000);
```

Here, **emp1** is a pointer to the memory where that Employee object is located. Then, if we wanted to keep an array of those employees, those pointers would refer to the original objects and not copies. As with the first approach, all that memory is relinquished when the program exits. However, it could be that recreating that list within some class would cause more memory to be reserved. This could eventually eat up a lot of memory. There are techniques for dealing with this that we'll take up later.

Right now, we simply want to create an Employees class that holds all the Employee objects. Inside the class, we'll use a

vector to hold the list of pointers. Note that the `addEmployee` method expects not an `Employee`, but a *pointer* to an `Employee`.

```
class Employees {
private:
    //contains an array of pointers to Employee objects
    vector <Employee *> employees;
public:
    //add a pointer to an Employee class to the vector
    void addEmployee(Employee* emp) {
        employees.push_back(emp);
    }
    //get the size of the vector
    int getCount() {
        return employees.size();
    }
    //get the pointer to the i-th Employee
    Employee* get(int i) {
        return employees[i];
    }
};
```

Finally our **main** method uses all these classes to create the list and print it out. Note that you can create the employee pointers one at a time as we illustrated above, or you can just create them within your call to the **addEmployee** method:

```
int main() {
    Employees employees;
    int id = 1;
    Employee* emp1 = new Employee(id++,
        "Susan", "Sugar", 5000);
    employees.addEmployee(emp1);
    employees.addEmployee(new Employee(id++,
        "Sarah", "Smythe", 2000));
    employees.addEmployee(new TempEmployee(id++,
        "Billy", "Bob", 1000));
    employees.addEmployee(new Intern(id++,
        "Arnold", "Stang", 800));

    for (int i=0; i< employees.getCount(); i++){
        Employee *emp = employees.get(i);
        cout << emp->getId()<<" " << emp->getName() <<
            " " << emp->getSalary() <<" "
```

```

        << emp->getBenefit() << endl;
    }
    return 0;
}

```

Classes and headers

In this simple little program, we actually created 4 classes:

- Employee
- TempEmployee
- Intern
- Employees

And all of them are dependent on the Employee class being defined first, since the Employees class *contains* instances of Employee and Intern and TempEmployee are derived from the base Employee class. Since these are relatively small, simple classes, it is convenient to put them all in the same file, and as long as the Employee class is defined first, everything will compile as expected.

But it is quite common to have situations where keeping the classes in that convenient order is much more difficult to achieve. In that case, it is very common to create prototypes of the classes at the top of the file, with the bodies of those classes below **main** as we did for our function prototype.

Let's take a look at the prototype for the Employee class:

```

class Employee {
protected:
    int idNum;
    string fname, lname;
    double salary;
    double benefits ;
public:
    Employee(int id, string frnm, string lnm,
             double sal,
             double benefits=1000);
    double getSalary();
}

```

```

    string getName();
    int getId();
    double getBenefit();
};

```

Note that all the public and private or protected methods are listed but terminated with a semicolon and no braces. Then the actual “guts” of the class are inserted below the main function:

```

//---Employee methods---
Employee::Employee(int id, string frnm, string
lnm, double sal,
                double benefits ) {
    fname = frnm;
    lname = lnm;
    idNum = id;
    salary = sal;
    this->benefits = benefits;
}
double Employee::getSalary() {return salary; }
string Employee::getName() {return fname + " "
+ lname; }
int Employee::getId() {return idNum; }
double Employee::getBenefit() {return
benefits;}

```

Here we see that each method is prefixed by the class name *and two colons*. Also, note that the default value for the **benefits** parameter is *not repeated*. It is only shown in the prototype. This makes sense, since the value must be known at compile time.

Here is the rather simple TempEmployee prototype:

```

class TempEmployee : public Employee {
public:
    TempEmployee(int id, string frnm, string lnm,
                double sal);
};

```

Note that the prototype does *not* include the inheritance relation to `Employee`, nor how the values are copied into the class. This takes place in the actual method section:

```
//---Temp Employee methods--
TempEmployee::TempEmployee(int id, string frnm,
    string lnm, double sal) :
    Employee(id, frnm, lnm, sal)
{
    benefits = 0;        //temps do not get benefits
}

```

Here the inheritance structure is shown, and the setting of the **benefits** parameter.

The **Intern** methods are shown in the prototype in a similar fashion. The private **capSalary** method is shown but the details are only shown in the method section:

```
class Intern : public Employee {
private:
    //cap the salary at 500 no matter what was entered
    void capSalary();

public:
    Intern(int id, string frnm, string lnm, double sal);
};

```

As before, the copying of values into the base class is only shown in the methods section.

```
//----Intern methods-----
    //cap the salary at 500 no matter what was entered
void Intern::capSalary() {
    if (salary > 500) {
        salary = 500;
    }
}
Intern::Intern(int id, string frnm, string lnm,
    double sal) :
    Employee(id, frnm, lnm, sal) {
    benefits = 0;        // no benefits either
    capSalary();        //cap the salary
}

```

The Employees class is similar.

Using Headers

All of the `#include` directives that start off C++ programs contain prototype-like files for what ever class they refer to. Those that are part of the standard C++ libraries have been precompiled for faster access, and those include files are written inside greater-than, less-than brackets like this

```
#include <string>
```

But for your own programs, you can make such include or *header* files for your own classes and they are exactly the same as the prototype files we wrote above. The only difference is that for each class, you make header file, such as “Employee.h” and a cpp file containing body of those methods. You have to include the appropriate namespace directives and any library include file declarations as well. Here is the header for our Employee class:

```
#ifndef EMPLOYEEHEADERS_EMPLOYEE_H
#define EMPLOYEEHEADERS_EMPLOYEE_H
#include <string>
using namespace std;

class Employee {
protected:
    int idNum;
    string fname, lname;
    double salary;
    double benefits ;
public:
    Employee(int id, string frnm, string lnm, double sal,
double benefits=1000);
    double getSalary();
    string getName();
    int getId();
    double getBenefit();
};

#endif //EMPLOYEEHEADERS_EMPLOYEE_H
```

And here is the body file Employee.cpp:

```
#include "Employee.h"
//---Employee methods---
Employee::Employee(int id, string frnm, string lnm, double
sal, double benefits ) {
    fname = frnm;
    lname = lnm;
    idNum = id;
    salary = sal;
    this->benefits = benefits;
}
double Employee::getSalary() {return salary; }
string Employee::getName() {return fname + " " + lname; }
int Employee::getId() {return idNum; }
double Employee::getBenefit() {return benefits;}
```

The interesting addition that IDEs like Visual Studio and CLion make is some sort of `#ifndef` statement to keep the compiler from reading the file in more than once if several others refer to it. In CLion, they look like the `Employee.h` above. In Visual Studio, it uses another approach by placing this single statement at the top of each include file:

```
#pragma once
```

The main program

Now that we've taken this relatively simple program apart, the main program is really just the code in `main` plus a bunch of includes. Here is the resulting main program:

```
#include "Employee.h"
#include "Employees.h"
#include "TempEmployee.h"
#include "Intern.h"

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



```

using std::string;
using std::vector;

//----Main program starts here----
int main() {
    Employees employees;
    int id = 1;
    Employee* emp1 = new Employee(id++,
        "Susan", "Sugar", 5000);
    employees.addEmployee(emp1);
    employees.addEmployee(new Employee(id++,
        "Sarah", "Smythe", 2000));
    employees.addEmployee(new TempEmployee(id++,
        "Billy", "Bob", 1000));
    employees.addEmployee(new Intern(id++,
        "Arnold", "Stang", 800));

    for (int i=0; i < employees.getCount(); i++){
        Employee *emp = employees.get(i);
        cout << emp->getId()<<" " << emp->getName() <<
            " " << emp->getSalary() <<" " << emp->getBenefit()
            << endl;
    }
    return 0;
}

```

Summary of headers

As long as you are writing short, experimental programs, you probably won't have to make prototype headers very often, and separate header files even less often. But you need to understand that most larger programs take advantage of these features to separate the various classes from each other.

Multiple Inheritance

Like Python, but unlike Java, for example, C++ supports *multiple* inheritance, where you can create a new class that has member methods (and variables) from two or more classes. As this can quickly get very tangled, this feature, you need to use it sparingly and thoughtfully. Most frequently, people create classes having multiple inheritance when they want to create a

class having methods that already exist in another class. Often this other class is more of an interface than a complex class, but once you have created such a class, you can treat as a member of either class hierarchy when convenient.

Let's take a really elementary example. Suppose you have a few employees that are really good at public speaking and represent your company well. You could derive a new class directly from `Employee`, or you could realize that you already have a `Speaker` class you could use:

```
class Speaker {
public:
    void inviteTalk() {
        cout << "Can you give a talk next week?" <<endl;
    }
    void giveTalk() {
        cout << "Greetings and blah blah blah..."<<endl;
    }
};
```

Then you could create `PublicEmployee` class by deriving it from both class hierarchies:

```
class PublicEmployee: public Employee, public Speaker {
public:
    PublicEmployee(int id, string frnm, string lnm, double
sal) :
        Employee(id, frnm, lnm, sal) {

    }
};
```

Here we create an instance in the usual way:

```
PublicEmployee* pemp = new PublicEmployee(id++,
    "Elizabeth", "Impressive", 6000);
pemp->inviteTalk();
```

And it is that easy to create a class with multiple inheritance. Note that you must make sure to declare the inheritance from both parent classes as **public** in order to use those public methods.

Polymorphism

Polymorphism is jaw breaking term for objects that change form. For example, our TempEmployee class changes form from the base Employee class by zeroing out the benefits value. This is essentially *method polymorphism*.

But another type of polymorphism, often called overloading is common in C++, and cannot easily be achieved in Python. To take a trivial example, suppose we made a class that has a method for adding two numbers together.

```
class Adder {
public:
    double addNums(double x, double y) {
        return x + y;
    }
};
```

Then, we could call it to carry out addition pretty simply:

```
Adder adder;    //create instance of Adder
cout << adder.addNums(12.1, 14) << endl;    //add 2 nums
```

But now, suppose we get string values from some visual entry field and wanted to add those. The method would have to invoke the string conversion function **stod** (string to double). Here's such a method:

```
//add numbers in two strings
double addNums(string x, string y) {
    return stod(x) + stod(y);
}
```

We would then call it by:

```
cout << adder.addNums("22.4", "1.008") << endl;
```

But note that this method has the same name and the same number of arguments: just different *types* of arguments. This is legal in C++ (although not directly in Python). And there could be two more for one string and one double as well:

```

double addNums(string x, double y) {
    return stod(x) + y;
}

double addNums(double x, string y) {
    return x + stod(y);
}

```

These are all legal and compile properly. In fact, we could change the *number* of arguments as well:

```

double addNums(double x, double y, double z){
    return x + y + + z;
}

```

and call it by

```

cout << adder.addNums(122.3,303.4,45.6) << endl;

```

This kind of overloading is common in C++, and as you can see, it can be pretty useful.

Virtual Functions

You can use the keyword **virtual** in C++ to indicate that functions are to be inherited in derived classes. Suppose we added the keyword to the **getSalary()** method in the Employee class above:

```

class Employee {
protected:
    int idNum;
    string fname, lname;
    double salary;
    double benefits;
public:
    Employee(int id, string frnm, string lnm,
             double sal, double benefits = 1000);
    virtual double getSalary();
    string getName();
    int getId();
    double getBenefit();
};

```

This indicates that there may be some derived classes that redefine that method in some way.

Then we could create a class called `Contractor` that returns some fraction of the total salary. (The rest might go to his agency.)

The definition of that class shows that there is a new `getSalary` method.

```
class Contractor:public Employee {
private:
    double rate = 0.85;
public:
    Contractor(int id, string frnm, string lnm,
               double sal, double benefits = 1000);
    double getSalary();
};
```

Then, the actual code for that method is somewhat different:

```
//reduce salary by "rate"
double Contractor::getSalary() {
    return salary * rate;
}
```

While that intent of **virtual** was to signal that certain functions may be redefined in derived classes, it is actually no longer necessary: that contractor code will work just fine without it in the current implementations of C++. The only difference is that if you declare a function to be virtual, its final implementation isn't resolved until run-time, which may make the program run slightly slower.

Pure Virtual Functions

You declare what is called a *pure virtual function* by following the declaration with a statement that it equals zero:

```
virtual double getSalary() = 0;
```

This statement means that this function does not exist in this base class but will be filled in in derived classes. In this case the base class is now *abstract* and cannot have any instances created.

In other words, this function is a abstract function that itself cannot be executed. You can't create an instance of a class with such an abstract function. You can only derive new classes from it that fill in that function. Those you can instantiate.

It's not likely that you'd create a abstract function and class out of our Employee class, because it doesn't do anything very useful. It is more likely that you use these pure virtual functions to create classes where all or nearly all of the functions are abstract. For example, in Chapter 13, we create a DButton class where the only function is abstract:

```
//an abstract button class
class DButton : public wxButton {
protected:
    Builder* bld;

public:
    DButton(...)    {
        this->bld = bld;
        Bind(wxEVT_BUTTON, &DButton::comd, this);
    }
//abstract method to be completed in derived classes
    virtual void comd(wxCommandEvent& event) =0;
};
```

And, in Chapter 23, we define the Bridger as an abstract class:

```
//abstract Bridge class
class Bridger {
    //add data to the other side of the bridge
    virtual void addData(Products* prod) = 0;
};
```

Static class members

This `Adder` class doesn't really have to be instantiated as we did in this example. We could make these methods **static** and call them directly.

```
class Adder {
public:
    //static methods do not need a class instance
    static double addNums(double x, double y) {
        return x + y;
    }
    static double addNums(string x, double y) {
        return stod(x) + y;
    }
    static double addNums(string x, string y) {
        return stod(x) + stod(y);
    }
    static double addNums(double x, string y) {
        return x + stod(y);
    }
    static double addNums(double x, double y, double z){
        return x + y + z;
    }
};
```

To call these methods, we don't have to create an instance of **Adder**. Instead, we use the *member-of* symbol (`::`).

```
cout << Adder::addNums(12.1, 14) << endl;
cout << Adder::addNums("22.4", "1.008") << endl;
cout << Adder::addNums(123.4, "6.02") << endl;
cout << Adder::addNums(122.3,303.4,45.6) << endl;
```

Note that these static class members do not have access to any instance variables the class may hold because they do not have a hidden or visible **this** pointer to reach them. Static class methods are useful for creating generally useful functions outside the class: they are not members of class instances.

Friend declarations

Early on in the design of C++, it seemed as though you might need to get at some of those private variables inside a class from time to time. But, as it turns out, you just don't need that feature. However, the **friend** declaration remains in the language. You can create a function outside a class and have that class declare that function as a friend. In that case the function can read and change those private variables. You can also declare a whole other class as a friend and it, too, will have access to the class that declared it a friend. This, of course, violates the whole idea of data encapsulation, and it would mean that such function or class would have to know exactly how data are represented in the friending class. We don't recommend this at all.

Constant classes

Suppose we consider our familiar Rectangle class for a moment:

```
class Rectangle {
private:
    double width;
    double height;
public:
    Rectangle(double w, double h):width{w}, height{h}{}
    void setWidth(double w){width = w;}
    void setHeight(double h) {height=h;    }
    double getWidth() {return width;}
    double getHeight() {return height;}
    double getArea() {return width * height;}
};
```

You can use the **const** modifier to change values that go in and out of this class. For example, you might want to make sure that the area is returned as a constant:

```
double getArea() const {return width * height;}
```

Like many other such modifications, they may not often be that significant.

But suppose you want to make the whole class constant!

```
const Rectangle rect{22,34};
```

Once the constructor has run and initialized the constant class, any attempt to modify the member variables will fail because they are now constant. But for this to work, all of the getter member functions must be labeled **const** as well, because the member private variables are const as well. So, for the output statement to work:

```
cout << rect.getArea()<<" " << rect.getWidth()<<
      " " <<rect.getHeight()<<endl;
```

the getters must all be constantized, and the setters might as well be removed because they can't work when the private variables are now constants. So, we have to modify our class as follows:

```
//void setWidth(double w){width = w;}
//void setHeight(double h) {height=h;    }
double getWidth() const {return width;}
double getHeight() const {return height;}
double getArea() const {return width * height;}
```

Example Programs

- textRectangle.cpp – draws rectangle and square with asterisks.
- Employee.cpp – Employee and derived classes
- EmployeeProto.cpp – Using prototypes for the classes.
- EmployeeHeaders folder – Using header files and separate class files
- PublicEmployee.cpp – illustrates multiple inheritance.
- addNumsPoly.cpp – 5 overloaded methods in Adder
- EmployeeVirtual.cpp
- PureVirtualEmployee.cpp

- staticPoly.cpp – 5 static methods in Adder
- constclass.cpp – constant Rectangle class

10. Pointers and Memory

This chapter deals with two ways to allocate memory: the old, common method used in C and early C++ and the newer smart pointer system that was introduced in C++ 11 (2011).

As we noted in the previous chapter, you can create space for new variables or arrays at compile time or at run time. In the first case, if you create an array at compile time:

```
double bigArray [1'000'000];    //huge static array
```

The space for this array is allocated on the *stack*, the same memory where computation takes place. This reserves a lot of memory for the whole program's execution time, when you may only need it for a little while. Note that we use the apostrophe as a digit separator to make it clear how large the number is. It has no computation effect.

Instead, it is better to create the memory array in free memory, often called the "heap."

```
//create a dynamic array on the heap
double* pbig {new double [DIM]{}};
```

Now you can assign values to this array, using it just like any other pointer:

```
double d =0;
for (size_t i=0; i< DIM; i++) {
    pbig[i] = d++; //convert to a double
}
```

But when you are done with this memory space, it is up to you to release it. This wouldn't matter in a small trial program that runs through and exits, because all memory will be released when the exit occurs. But in many C++ and C programs request memory and forget to release it, leading to a gradual increase in memory usage until the system could run out of memory.

So, it is up to you to use the **delete** method to release any memory you request. In this example, you would release that `pbig` array with

```
//release double array
delete [] pbig;
```

You could also create your memory using a vector when you need it.

```
//create a dynamic array
vector <double *> dubbles;
for (size_t i=0; i< 1'000'000; i++) {
    double* px = new double;
    dubbles.push_back(px);
}
```

Here we actually are creating a set of *pointers* to **double** values, as you can see when we print some out.

```
for (size_t k=0; k<5; k++) {
    cout << k<<" "<< *dubbles[k]<<endl;
}
```

In this case, `dubbles[k]` is a pointer to the double precision variable, so to print out that values, we print `*dubbles[k]`.

Then, to release all those individual memory reservations, we have to run through the whole vector and delete them:

```
//release memory from dynamic array
for (double* db:dubbles){
    delete db;
}
```

Classes and destructors

However, it is more common that you might need to keep track of memory and manage it within classes. We have already seen

in detail how class constructors work. They initialize variables and structures within each instance of a class.

However, when the program is done with a class, it calls that class's *destructor*. The destructor has the same name as the class, but prefixed with a tilde (~). Again, if you have a short program that runs through some code once and exits, you really don't care. All that memory will be released anyway.

But if you have several classes that acquire memory and should release them when they are done, you need to provide a class destructor method. If you recall our Employees class from the previous chapter, it contains a vector, which itself contains an array of pointer to Employee classes. You need to release all that memory in the destructor. Here is that Employees class showing that destructor:

```
class Employees {
private:
    //contains an array of pointers to Employee objects
    vector <Employee *> employees;
public:
    //add a pointer to an Employee class to the vector
    void addEmployee(Employee* emp) {
        employees.push_back(emp);
    }
    //destructor releases memory
    ~Employees() {
        for (Employee* emp:employees) {
            delete emp;        //release each instance
        }
    }
};
```

When is the destructor called?

The destructor is called whenever a class goes “out of scope,” so no one could use it further. If you create a class and use it inside a pair of braces, as soon as the program execution goes outside those braces (usually a function method), the destructor is called automatically.

For example,

```
void mkadd() {
    Employees empls;
    empls.add(...);
}
```

Outside of those braces, **empls** does not exist, so the Employees destructor is called and you need to delete any temporary memory you may have acquired. If you don't you may end up with memory leaks.

Other uses for destructors

Any class that acquires system resources should release them when the destructor is called. Obvious examples includes files. If you open a file, the destructor should close it. If you have created a temporary file, you should probably delete it here.

Smart Pointers

Smart pointers were added in 2011 and they manage themselves: you don't have to release any memory you allocate. These are called **unique_ptr**'s and you can easily create them anywhere you would create the older C-type pointers:

```
unique_ptr <Employee> emp1 =
    make_unique <Employee>(id++, "Susan", "Sugar", 5000);
```

Essentially, this says to create a unique pointer to a block of memory where it stores an instance of the Employee class. There can only be one instance of each of these unique pointers, and you can't copy them as part of a function call. For example, if **empl** were an ordinary pointer, you could add it to a vector like this:

```
employees.push_back(empl);
```

But since that would create a copy of the pointer, the compiler won't allow you to do this. You have two ways that do work, though.

First, you could pass the actual class instance into the `Employees` class and have it create the unique pointer:

```
void addEmployee(Employee emp) {
    employees.push_back(make_unique<Employee> (emp));
}
```

Or, you could create the pointer in the calling program as we did above and tell the compiler that you are going to **move** it into the vector.

```
void addEmployee(unique_ptr<Employee> emp) {
    employees.push_back(std::move(emp));
}
```

That way, there is no copying, and this works just fine. Either way, when you later fetch that pointer to get the `Employee` instance, it works just like an ordinary pointer:

```
//get the pointer to the i-th Employee
//and return the actual Employee instance
Employee get(int i) {
    Employee emp = *employees[i];
    return emp;
}
```

While getting used to not accidentally copying pointers takes a bit of time, this is a far safer way to write bigger programs and avoid memory leaks.

Example Code on GitHub

- `Destruct.cpp` – shows creating pointers and destroying them
- `EmployeePtr.cpp` – passes a class instance in to avoid copying a unique pointer
- `EmployeeUniquMove` – shows how to move a unique pointer

11. Using linked lists

Linked lists are an important part of building useful software projects. While you can consider them as a kind of arrays, they are considerably more versatile than that. You can use them to represent sparse arrays or matrices, cells in a spreadsheet, lists of commands to be executed or even lists of open windows in a user interface. Just as important, it is very easy to insert or delete members of a list without a lot of memory manipulation.

Definitions

A linked list is simply a linear chain of elements, or *nodes*. Nodes are usually represented by instances of a C++ class. One node is called the *head* and forms the beginning of the list. Not surprisingly, the last element is generally called the *tail*. Each node consists of a pointer to the next node in the list, starting with the head and continuing up to the tail. Such a list is called a *singly linked* list as shown in the Figure below.



In a *doubly linked* list, there are also pointers to the previous element, so you can traverse the list in either direction.



Usually, the last element has a next point with a Null value. Linked lists are also possible in Python, although they use references rather than pointers and may not be as fast.

In our linked list, we create a Cell class that holds those two pointers as well as a pointer to some kind of data class. In this example, we'll just use our same old Employee class:

```
class Cell {
private:
    Employee* data;
    Cell* next{NULL};
    Cell* prev{NULL};
public:
    //constructor uses Employee pointer
    Cell(Employee* emp):data{emp}{}
    Employee* getData() {return data;} //return employee
//add a cell to the end of the chain
void addNext(Cell* ecell) {
    next = ecell;
}
}
```

The linked list class itself manages the head and tail pointers and has the methods for adding cells to the list:

```
//constructs the Linked List
class LinkedList{
private:
    Cell* head;    //start of List
    Cell* tail;    //Last member of List
public:
    LinkedList(Cell* ecell){
        head = ecell;
        tail = ecell;
    }
//add cell to end of List
void addCell (Cell* ecell) {
    tail->addNext(ecell);
    auto oldtail = tail;
    tail = ecell;
    ecell->addPrev(oldtail);
}
}
```

As you can see from the **addCell** method, it calls the Cell's **addNext** method that adds one more cell to the tail of the chain.

Creating the list

You create the linked list, by creating Cells and adding them to the LinkedList object. First we create the Employee pointer:

```
int id = 1;
//create new Employee pointer
Employee* emp1 = new Employee(id++,
    "Susan", "Sugar", 5000);
```

Then we create a new Cell for it:

```
//create a new Cell
Cell* cell1 = new Cell(emp1);
```

And finally, we use that Cell pointer to create the LinkedList class:

```
//create the linked list with one cell in it.
LinkedList* links = new LinkedList(cell1);
```

We can create and add the rest of the cells in single statements:

```
//create remaining Employees and Cells in one statement
links->addCell(new Cell(new Employee(id++,
    "Sarah", "Smythe", 2000)));
links->addCell(new Cell(new Employee(id++,
    "Billy", "Bob", 1000)));
links->addCell(new Cell(new Employee(id++,
    "Arnold", "Stang", 800)));
```

Traversing the list

Then, it is really easy to move through a linked list: each cell has a pointer to the next cell in the chain, and the last cell points to NULL, indicating that you are done. In order to keep that code from being part of the main program, the LinkList class returns

an iterator to the list. That iterator, like those suggested in *Design Patterns* has only a **hasMore** and a **getNext** method. We will discuss the C++ iterator style later.

But to run through the list you need only get the iterator from the `LinkedList` and use it:

```
fwdIter* fwd = links->getFwdIter(); //get the iterator
while (fwd->hasMore()) {           //get the elements
    Cell* cell = fwd->getNext();
    Employee* emp = cell->getData(); //get data in cell
    cout << emp->getName() << endl; //and print it.
}
```

The **fwdIter** class checks the cells to see if you have reached the end:

```
class fwdIter{
protected:
    Cell* cell;

public:
    fwdIter(Cell* c) {
        cell = c; //save the starting cell
    }
    bool hasMore() {
        return cell != NULL; //no more if NULL
    }
    Cell* getNext() {
        auto retCell = cell; //save this cell
        cell = cell->getNext(); //get the next(or NULL)
        return retCell; //return current cell
    }
};
```

The tricky part of this little iterator is that it returns the *current* cell and then fetches the next one, which may be `NULL`. Then when the program next calls **hasMore**, it can return **true** as long as the new current cell is not `NULL`.

The reverse iterator

You can derive the reverse iterator from the forward iterator. This derived class only contains a new **getNext** method: the rest is the same:

```
//iterator to move backward from end of list
class revIter: public fwdIter {
public:
    revIter(Cell* c) :fwdIter(c){
    }
    Cell* getNext() {
        auto retCell = cell;
        cell = cell->getPrev(); //get the previous cell
        return retCell;
    }
};
```

So, to print out the linked list from back to front, we get the reverse iterator from the LinkedList and use it just like the forward one:

```
revIter* rev = links->getRevIter();
while (rev->hasMore()) {
    Cell* cell = rev->getNext(); //get the prior cell
    Employee* emp = cell->getData(); //and its data
    cout << emp->getName() << endl; //and print it out
}
```

Inserting a new cell in the chain

You can insert a cell without moving any arrays around when you are using linked lists. You just have to switch the pointers so that the old left cell points to the new cell and the new cell points to the right cell. This **insertCell** method is part of our LinkedList class:

```
//insert cell after cleft
void insertCell(Cell* cleft, Cell* cnew) {
    Cell* cright = cleft->getNext(); //cell to right
    cleft->setNext(cnew); //set pointer to new cell
    cnew->setNext(cright); //set right pointer in cnew
}
```

```

    cright->setPrev(cnew); //set prev pointer in cright
    cnew->setPrev(cleft); //set prev pointer in cnew
}

```

The copy constructor

Now, suppose you want to create a new instance of a Cell class. We could use one of the ones above and use it to make a new one:

```

//illustrates copy constructor
Cell c = *cell12; //get a cell from above
Cell newCell = Cell(c); //use copy cons. to make new cell

```

So, what exactly is in this **newCell**? It turns out that this heretofore unmentioned constructor copies all of the variables inside that instance of the class into the new one. Let's try this out:

```

Employee* cdat = newCell.getData();
cout << newCell.getData()->getName() << endl;

```

Amazingly, this works just fine, and in this case prints out the name “Bonnie Ocean.” (Her middle name must be Lyzoverthe!) So, this means that all the pointers in that first cell are copied into the new cell. This might not be such a great plan, because those pointers might very well get deleted by a destructor, leaving this new cell with one or more invalid pointers.

Every C++ class has a hidden copy constructor, and all it does is copy all the fields, whether you want that or not. If such a copied class might have such pointers lying about, you can override the copy constructor and set them to NULL like this:

```

//copy constructor
Cell(Cell &cnew) {
    data = NULL;
    next = NULL;
    prev = NULL;
}

```

The syntax of a copy constructor is just the class name and a reference to the new cell name. Here is where you could null out those pointers to prevent them being used where they may fail.

Is this trip really necessary?

Well, “this is all very well,” you might say, “but I never use copy constructors.” Well C++ uses them under the covers a lot, so don’t be too sure.

Suppose you want to pass an instance of the Cell class to a function. (Not a pointer, now, a reference to the actual function.) You might want to carry out some operation there in that function. But for now, we’ll just return the name of the Employee:

```
string names(Cell c) {
    Employee* emp = c.getData(); //ptr to Employee
    Employee e = *emp;           //actual Employee
    nm = e.getName();           //get the name
    return nm;                  //and return it.
}
```

This looks like it should work, and it will if you haven’t modified the copy constructor as we did above, because such function calls *copy the class instance*. And of course, they use the copy constructor! If you did modify the copy constructor, the Employee pointer will be NULL and you won’t get any value for names. Here is an example of how you might handle that:

```
string names(Cell c) {
    string nm = "no data";
    Employee* emp = c.getData();
    if (emp != NULL) {
        Employee e = *emp;
        nm = e.getName();
    }
    return nm;
}
```

However, if you have modified the copy constructor, **emp** will always be null.

So, the copy constructor can get you even when you aren't looking for it. Of course, the simplest way around this is to use pointers to the classes, and everything will work, since no copy constructor is ever called.

Deleting the copy constructor

One way to make sure this does not happen is to delete the copy constructor instead of modifying it. Here is how to do this for the Cell object:

```
Cell(Cell &cnew) = delete;
```

Summary

Linked lists are an extremely efficient way of organizing sparse lists of data. You can run through them sequentially very rapidly and it is fast and easy to insert or delete an element without moving anything around in memory. The only disadvantage is that searching them is not terribly efficient. And, of course, it is up to you to manage the pointers and memory that you allocate. You also have to be careful of the cell object you use to contain list elements and be sure that you don't misuse copy constructors in the process.

Example Code

- `LinkedList.cpp` -- builds a doubly linked list of Employees
- `Copycon.cpp` – shows copy constructor and how it could fail

12. Templates

Templates don't have any close analogy in Python: they are pretty much unique to C++. Essentially templates are a special kind of macro that allows you to write functions and classes without requiring a specific type of data. Instead, you create a template type which the compiler fills in, generating the actual code for each type you require. Templates are deeply entrenched in the C++ Standard Template Library, and you sometimes use them without even realizing it.

Template functions

For example, the `swap` function is really a template function, because you can use it to swap two variables of any type. Here we swap two doubles:

```
double a =123;
double b = 456;
swap(a,b);
cout << "a="<< a <<" b=" << b << endl;
```

and here we swap two strings:

```
string fruit1 = "banana";
string fruit2 = "orange";
swap(fruit1, fruit2);
cout << "fruit1="<< fruit1 <<" fruit2="<<fruit2<<endl;
```

The result of this little program is, of course:

```
a=456 b=123
fruit1=orange fruit2=banana
```

What is going on under the covers is that the `swap` function is really defined as a simple template. We name it `mySwap` so it doesn't collide with the existing `std::swap` template.

```
//our own swap template function
template <typename myType>
void mySwap(myType a, myType b) {
    myType temp = a;
    a = b;
    b = temp;
}
```

It is common in C++ programs to just use the symbol **T** for the type variable. Sometimes those who are new to C++ find this confusing, so we started with **myType** above, but the actual code is generally more like:

```
//our own swap template function
template <typename T>
void mySwap(T a, T b) {
    T temp = a;
    a = b;
    b = temp;
}
```

You can use any type name here that you like, but **T** is commonly used.

Class templates

Template functions occur frequently in C++'s template library but are used somewhat less frequently in user code than classes that utilize templates. So, extending an example in TutorialsPoint.com, let's consider how we might build a Stack class.

Stacks are entities much like vectors, except that they are mostly used to push on and pop off values. So, we will use a class template to create a stack from a vector. When we push a value onto the stack, it is the top item, and the first one to be removed. Unlike the vector class's behavior our pop operation actually returns that value. We really only need to find out if there are any remaining values on the stack, and for this we'll create a **hasValues()** method which is simply the negation of the vector's **empty()** method.

When you create a template class, you start by declaring the place holders for the types it will use. In this case, there is only one:

```
template <class T>
```

Note that while we commonly use the keyword **class** here, you could just as well have used **typename**. They are interchangeable. Our class is very simple, being made up of just the three methods:

```
push()
pop(), and
hasValues()
```

The class starts as shown below:

```
class Stack {
private:
    vector<T> values;    //type specified here
};
```

The vector is of type **T**, where **T** can be any simple type or any class instance. The push and hasData methods are equally simple:

```
public:
    void push(T val){    //push onto stack
        values.push_back(val);
    }

    //return true if there are any values left
    bool hasValues() {
        return !values.empty();
    }
}
```

The only real work we have to do is to write a form of **pop** that returns the top element rather than discarding it:

```
//pop a value from the stack
T pop() {
    if (hasValues()) {
        //save last value
        T popval = values[values.size()-1];
```

```

        values.pop_back(); //and remove it
        return popval;
    }
    else
        return NULL;
}

```

Note that the **pop** method returns a value of type **T**. Our calling program is very simple. We simply specify the type the Stack will handle and everything else is the same as usual:

```

int main() {
    Stack<int> stack; //create stack
    stack.push(20); //push on 3 values
    stack.push(42);
    stack.push(91);

    //pop off one at a time and print it
    do {
        cout << stack.pop() << endl;
    } while (stack.hasValues());

    return 0;
}

```

The resulting output, is the list of numbers in reverse order as they come off the stack:

```

91
42
20

```

Class templates of classes

Of course, you aren't limited to simple types in creating templates: you can use any class instances you want. For a simple, but trivial, example let's consider an area calculating template class called **DoArea**. It can take any class which has a **getArea()** method.

So here is our Rectangle class, much like ones we've written before.

```

//Rectangle class
class Rectangle {
private:
    double width;
    double height;
public:
    //constructor
    Rectangle(double w, double h):width{w},height{h}{
    }
    //default constructor
    Rectangle(){}
    double area() {
        return width * height;
    }
};

```

It is important to note that classes to be used in templates *must* have a default constructor: that is a constructor that has no arguments like the **Rectangle()** constructor above.

Similarly, we can write a **Semicircle** class that returns its area: $\pi r^2/2$.

```

class Semicircle {
private:
    double radius=0;
public:
    //constructor
    Semicircle(double rad):radius{rad} {
    }
    //default constructor
    Semicircle(){}
    double area() {
        return radius * radius * pi/2;
    }
};

```

The math constant π is available in C++ version 20 along with quite a lot of others, as long as you include the following in the program header.

```

#include <numbers>
using namespace std::numbers;

```

A link to the complete list of math symbols is given in the References below.

With those classes in mind, here's how we build a template to return areas:

```
//gets the area of any class
//with a getArea method
template <class T>
class DoArea {
private:
    T shape ;
public:
    DoArea (T tshape ):shape{tshape} {
    }
    double getArea() {
        return shape.area();
    }
};
```

So, to call this template class method we create instances of the two classes:

```
//create 2 shapes
Rectangle rect = Rectangle(5,6);
Semicircle semi = Semicircle(7);
```

Then we can use the template to get the area:

```
//get area of semicircle
DoArea doarea = DoArea(semi);
cout << "Semicircle " << doarea.getArea() << endl;
```

Or, in a single statement:

```
//get area of rectangle
cout << "Rectangle " << DoArea(rect).getArea() << endl;
```

References

1. <https://www.tutorialspoint.com/cplusplus/>
2. [cpp_templates.htm](#)
3. https://en.cppreference.com/w/cpp/symbol_index/numbers

Example Code

- Swapper.cpp – swap function
- Stack.cpp – creates stack from vector
- shapes.cpp – Template for area of rectangle and semicircle

13. Creating user interfaces

In Python, you can create nice looking user interfaces using the provided **tkinter** toolkit, or one of the external products like PyQt or wxPython. Of these, only wxPython has a C++ equivalent, because they are built on the same code base. If you have looked at wxPython, you will find that it is much like tkinter in the objects it creates and the layout managers it uses.

The C++ version, called **wxWidgets**, is well designed and easy to use, and gives you a way to create buttons, labels, listboxes, tables and entry fields, and interact with the user. While wxWidgets can run on all three major platforms: Windows, Macintosh and Linux, it works best in Windows using Visual Studio Community Edition. Installation of wxWidgets amounts to downloading code libraries and setting a number of environment variables. We describe this installation at the end of this chapter. While it has been reported that you can also install it to use the CLion IDE, we haven't tried it. However, there is a reference to that article at the end of the chapter.

Most documentation for wxWidgets is online, and you can usually find the answer to how to do something pretty quickly. Note that there is a web site containing a complete description of every widget in the package, and all of each widget's methods, but with little or no example code. A search for the widget name followed by "example code" will usually give you what you want. Two of the authors of wxWidgets wrote a book on the system in 2008, which is still available for around \$50, although used copies are also available. The problem is that the wxWidgets system has evolved significantly since that yeoman chore was completed, notably in event handling, and the book is no longer that useful.

A wxPython example

Let's start by looking at the rather simple code wxPython requires to display a window with a title in the title bar. Like tkinter, you start by setting up the window and then launching the window event system by calling the **app.MainLoop()** method.

```
# Import the wxPython package.
import wx

app = wx.App() # Create an application object.
frm = wx.Frame(None, title="Hello World") # Then a frame.
frm.SetInitialSize((250, 200)) #set the size
frm.Show() # Show it.

# Start the event loop.
app.MainLoop()
```

This will create a simple 250 x 200 pixel window with a title in the title bar.

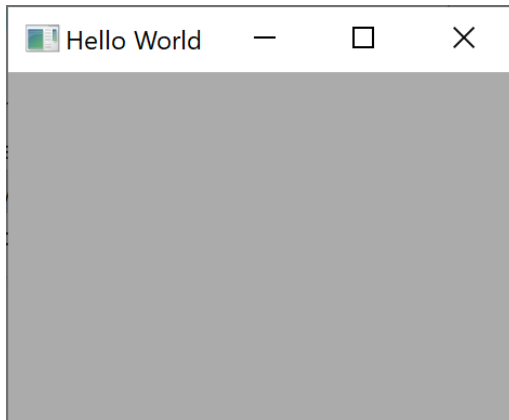


Figure 13-1 - Python window with title bar

You can launch a similar window in C++ using the wxWidgets toolkit:

```

#ifndef WX_PRECOMP
#include <wx/wx.h>
#include "wx/app.h"
#endif

//create the app
class MyApp : public wxApp
{
public:
    bool OnInit() {          //called to start the UI
        wxFrame* frame =    //create the frame
            new wxFrame(NULL, wxID_ANY, "Hello World");

        frame->SetSize(250, 200); //set a size
        frame->Show(true);        //and show it
        return true;
    }
};
//launch the app
wxIMPLEMENT_APP(MyApp);

```

Note that the very last line is a macro that actually starts the window and event code running. The identical window this code generates is shown in Figure 13-2.

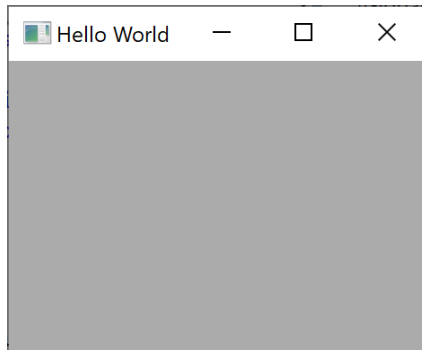


Figure 13-2 --wxWidgets window with title bar

In both cases we create a frame with a title for the title bar, set the size and launch the app. In the C++ case, you create an app

derived from `wxApp`, that has an `OnInit` method that gets called from the `wxIMPLEMENT_APP` macro. This kicks off the event processing code.

Strings in wxWidgets

Nearly all of the methods in the `wxWidgets` objects require that you call them with a string converted to a `wxString` object. The `wxString` behaves mostly like any other string, but it can handle Unicode characters as well. So, you will see calls to many methods converting your string to a `wxString` like this one:

```
wxStaticText* tx = new wxStaticText(panel, wxID_ANY,
    wxString("Greetings"), wxPoint(40,60));
```

There is also a shorter spelling of this method as a macro called `wxT()`:

```
wxStaticText* tx = new wxStaticText(panel, wxID_ANY,
    wxT("Greetings"), wxPoint(40,60));
```

They work the same way. Likewise, reading entry fields returns a `wxString` rather than a C++ string, but these widgets provide a conversion method: `ToStdString()`, like this one:

```
string st1 = num1->GetLineText(0).ToStdString();
```

Writing basic wxWidgets code

As you can see, creating a window amounts to creating a frame. But, if we want to put components inside that frame, we have to create a `wxPanel` as well. While you can place some widgets directly in a `wxFrame`, you can't position them at all. And further, if they are inside a `wxPanel`, you can tab between them using your keyboard's Tab key.

So, in this next simple example, we create a panel and put a label inside it. Note, however, that unlike other GUI systems, labels

are called **wxStaticText** objects. So, we create a frame, put a panel inside it and place the label inside that.

```
class MyApp : public wxApp {
public:
    bool OnInit() {
        wxFrame* frame = new wxFrame(NULL, wxID_ANY,
                                     "Hello World");
        frame->SetSize(250, 200);
        wxPanel* panel = new wxPanel(frame);

        wxStaticText* tx = new wxStaticText(panel,
                                             wxID_ANY,
                                             wxString("Greetings"),
                                             wxPoint(40,60));

        frame->Show(true);
        return true;
    }
};
```

Note that we don't "add" the label to the panel. We just say that the parent window of the **wxStaticText** control is the panel. Note also, that in this first such case, we specify the coordinates of that label right in the constructor, as a **wxPoint** object, with the(40, 60) coordinates specified. The resulting window looks like this:

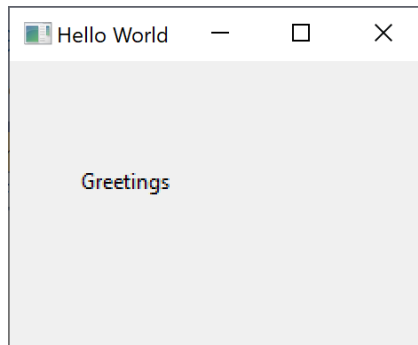


Figure 13-3 --Static text positioned at (40,60)

With the “Greetings: label not at (40,60).

Sizers

In Python’s tkinter, you can arrange your visual objects using Layout Managers. In wxWidgets, these are called *sizers*, but have much the same functions. Sizers are pretty simple to use, and there are just a few of them.

- wxBoxSizer – a vertical or horizontal layout with one object per row or column.
- wxStaticBoxSizer – includes a labelled frame around the box region.
- wxGridSizer – evenly sized grid rows and columns
- wxFlexGridSizer – sizes or grid cells are adjusted to fit your widgets
- wxGridBagSizer – flexible grid size and you can specify the grid position directly.

Include Files

Most of the wxWidgets have their own include files, all under the wx directory:

```
#include "wx/wx.h" - for base objects
#include "wx/app.h" - for all apps
#include "wx/button.h" - for buttons
#include "wx/sizer.h" - for most of the sizers
#include "wx/gbsizer.h" - for the GridBagSizer
#include "wx/treectrl.h" - for the TreeCtrl
```

The Box Sizer

The most common layout tool is the wxBoxSizer. You generally create a panel and then add a sizer to it like this:

```
//create the Box sizer
wxBoxSizer* vbox = new wxBoxSizer(wxVERTICAL);
panel->SetSizer(vbox);
```

With the BoxSizer, you can select either vertical or horizontal orientation. Each item you add to the sizer is thus either in a new row (`wxVERTICAL`) or a new column (`wxHORIZONTAL`). You can begin right at the top for the vertical sizer, or you can add some space first:

```
//create the text label
    wxStaticText* tx = new wxStaticText(panel, wxID_ANY,
        wxString("Greetings"));
    vbox->AddSpacer(50);
    vbox->Add(tx);    //and add it to the box
```

or, you can add the widget with center or right positioning. For example,

```
vbox->Add(tx,
    0,        //not stretchable
    wxALIGN_CENTER, //alignment
    10);     //border width
```

These are both shown in .

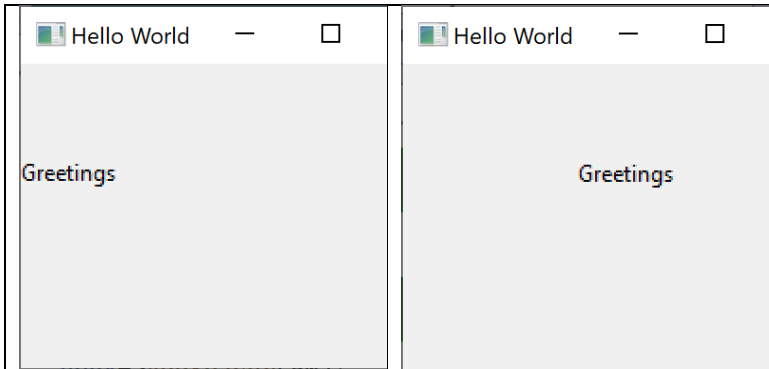


Figure 13-4 Box sizer without(left) and with(right) `wxALIGN_CENTER`

Splitting up the main app

While the basic example code provided with `wxWidgets` uses `MyApp` as the name of the app that launches the program, we usually call it `Builder`. Then, we recommend splitting up the class to a header declaration.

```
class Builder : public wxApp {
public:
    bool OnInit();
};
```

And putting the code for OnInit below the

```
wxIMPLEMENT_APP(Builder);
```

In this case, that code simply builds the window just as we did above:

```
bool Builder::OnInit() {
    wxFrame* frame = new wxFrame(NULL, wxID_ANY,
        "Hello World");
    frame->SetSize(250, 200);
    wxPanel* panel = new wxPanel(frame);

    //create the Box sizer
    wxBoxSizer* vbox = new wxBoxSizer(wxVERTICAL);
    panel->SetSizer(vbox);

    //create the text label
    wxStaticText* tx = new wxStaticText(panel, wxID_ANY,
        wxString("Greetings"));
    vbox->AddSpacer(50);
    vbox->Add(tx);    //and add it to the box
    frame->Show(true);
    return true;
}
```

More on labels

The text size and color are adjustable, of course, and we have found it convenient to create a derived BlueLabel class we can use throughout. In this class, we make the color blue and the font size a bit bigger, 12 point instead of the default 10 point.

Colors in wxWidgets can be represented as 3 integers between 0 and 255 for the red, green and blue base colors. So, to change the color of a label, you could write:

```
SetForegroundColour(wxColour(0, 0, 200));
```


Or for most common colors, you can just use a quoted string

```
SetForegroundColour("blue");
```

Let's create our BlueLabel class as a header and a body that we can put in any project we want. The header is:

```
class BlueLabel : public wxStaticText {
public:
    BlueLabel(wxPanel* parent, int id,
              const wxString& label);
};
```

And the body in the cpp file is just:

```
//----A derived class for blue labels-----
BlueLabel::BlueLabel(wxPanel* parent, int id,
    const wxString& label) :
    wxStaticText(parent, id, label) {
    SetForegroundColour("blue");
    wxFont font = wxFont(10,
        wxFONTFAMILY_DEFAULT, wxFONTSTYLE_NORMAL,
        wxFONTWEIGHT_NORMAL, FALSE, "");
    this->SetFont(font);
}
```

Entry fields and buttons

Now, let's write just a slightly more complicated window that allows you to enter your name, click on a button and have your name echoed back to you.

Our new window will have 4 lines in a vertical BoxSizer:

- a title BlueLabel
- an entry field
- a "Say hi" button
- a BlueLabel where the greeting is displayed

In this program entry fields are called `wxTextCtrl` widgets, and buttons are called `wxButton` widgets.

So setting up the layout should be very simple. First, we declare this widgets as class instance variables:

```
class Builder : public wxApp {
private:
    BlueLabel* title;
    wxTextCtrl* name;
    wxButton* butn;
    BlueLabel* greeting;
```

And then, in the Builder `OnInit()` method, we place them in a vertical `BoxSizer`:

```
vbox->AddSpacer(20);
vbox->Add(title, 0, wxALIGN_CENTER, 10);

name = new wxTextCtrl(panel, wxID_ANY);
vbox->Add(name, 0, wxALIGN_CENTER, 10);

vbox->AddSpacer(10);
butn = new wxButton(panel, wxID_ANY, "Say hi");
vbox->Add(butn, 0, wxALIGN_CENTER, 10);

vbox->AddSpacer(10);
greeting = new BlueLabel(panel,
                        wxID_ANY, "");
vbox->Add(greeting, 0, wxALIGN_CENTER, 10);
```

Note that the message label at the bottom is filled with blanks so it will more or less stay centered for various length names. The window we have created looks like Figure 13-5.

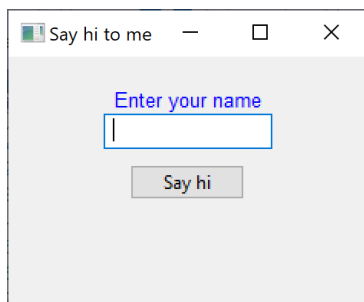


Figure 13-5 Entry field, push button and blank label

But what about the button? What does it do and how do we handle it?

Events in wxWidgets

Every operation that causes a change in a window object generates an *event*. The obvious ones are button clicks, listbox clicks, checkbox clicks and so forth. But there are also events when the text changes in an entry field, or when scrollbars move, and so forth. Every event inherits from the base **wxEvt** class, and each such event contains a pointer and ID of the widget that generated the event, so you can distinguish identical events from different sources. Note that in scrolling through the copious online wxWidgets documentation, you will also find references to the older event table approach. These tables are constructed at compile time, while the later Bind event handling is more flexible as it can be changed while the program is running. We will discuss only the later, and more flexible, Bind method.

For each event you want to intercept, you must bind the event to a method in an existing instance of the class. To simplify this, you usually bind to a method in the Builder class, and let that method call other classes if it needs to.

For our single “Say hi” button above this amounts to issuing a single Bind call like this one:

```
//Now add in button click event
    butn->Bind(wxEVT_BUTTON, &Builder::OnClick, this);
```

What this Bind call does is that if the wxButton named **butn** issues a **wxEVT_BUTTON** event, then that button click should call the **OnClick** method in the **Builder** class.

Of course, we have to declare the OnClick method in the header section:

```
public:
    bool OnInit();
    void OnClick(wxCommandEvent& event);
```

And in the actual OnClick method, we just fetch the text string from the name field and prepend a “Hi” to it and put it in the greeting label.

```
void Builder::OnClick(wxCommandEvent& event) {
    //get the text and convert it to a string
    string text = name->GetLabelText(0).StdString();
    string grtext = "Hi " + text; //prepend "Hi"
    if (text == "Jim") grtext += " boy!";
    //put result in greeting
    greeting->SetLabelText(grtext);
}
```

In honor of Robert Heinlein, if the name is “Jim” you get a special revised greeting.

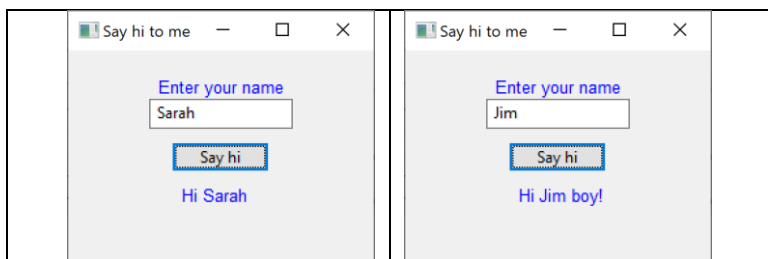


Figure 13-6 - Greeting to Sarah and to Jim

Adding two numbers together

This example appears at first to be quite similar to the previous one, However, we will use it to show you several new concepts:

- The GridBag sizer
- Buttons and the Command design pattern
- Formatting numbers in a label
- A virtual function

Here is the user interface we have constructed:

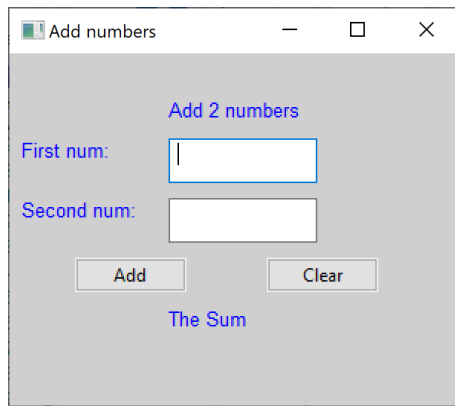


Figure 13-7 - Add 2 numbers using GridBag layout sizer

The GridBag Sizer

You create a GridBag sizer as a grid of rows and columns. It doesn't matter if you specify more rows and columns than you end up using: only ones that are populated will show in the window. Here we first create a 10x10 grid but only end up using 6 in each direction:

```
wxGridBagSizer* gbs = new wxGridBagSizer(10, 10);
panel->SetSizer(gbs);
```

Now, our grid looks like this:

	0	1	2	3
0				
1		Add 2 numbers		
2	First num	<i>Entry field num1</i>		
3	Second num	<i>Entry field num2</i>		
4	Add		Clear	
5		The Sum		

Note that the text in row 1 begins in column 1. We achieve this using the `wxGBSpan` method that specifies the starting row and the number of columns:

```
BlueLabel* topTitle =
    new BlueLabel(panel, wxID_ANY,
        wxT("Add 2 numbers"));
gbs->Add(topTitle,
    wxGBPosition(1, 1), wxGBSpan(1, 3));
```

The same approach applies to the two rows of labels and entry fields:

```
//first label and entry field
BlueLabel* lineLabel =
    new BlueLabel(panel, wxID_ANY, " First num: ");
gbs->Add(lineLabel, wxGBPosition(2, 0));

num1 = new wxTextCtrl(panel, wxID_ANY, "",
    wxDefaultPosition, wxSize(100, 30));
gbs->Add(this->num1, wxGBPosition(2, 1),
    wxGBSpan(1, 2));

//second label and entry field
BlueLabel* lineLabel2 =
    new BlueLabel(panel, wxID_ANY,
        " Second num: ");
gbs->Add(lineLabel2, wxGBPosition(3, 0));

num2 = new wxTextCtrl(panel, wxID_ANY, "",
    wxDefaultPosition, wxSize(100, 30));
gbs->Add(num2, wxGBPosition(3, 1), wxGBSpan(1, 2));
```

The Add and Clear buttons are each centered in two columns: 0-1 and 2-3:

```
// Add button
AddButton* addButton = new AddButton(panel,
                                     wxID_ANY, this, this);
gbs->Add(addButton, wxGBPosition(4, 0),
        wxGBSpan(1, 2), wxALIGN_CENTER_HORIZONTAL);

// Clear button
ClearButton* clearButton =
    new ClearButton(panel, wxID_ANY, this, this);

wxSizerItem* obj = gbs->Add(clearButton,
                           wxGBPosition(4, 2),
                           wxGBSpan(1, 2),
                           wxALIGN_LEFT);
```

And, finally, the Sum label at the bottom begins in column 1, with nothing in column 0.

```
// sum label
sumLbl = new BlueLabel(panel, wxID_ANY, "The Sum ");
gbs->Add(sumLbl, wxGBPosition(5, 1));
```

The Add and Clear buttons

But what are those Add and Clear buttons? They clearly are derived from `wxButton`, but why did we do that? Let's start with the code in the Builder that carries out the Addition and the Clearing of the form:

```
// This is the Add button click event
void Builder::addClicked(wxCommandEvent& event) {
    string st1 = num1->GetLineText(0).ToStdString();
    string st2 = num2->GetLineText(0).ToStdString();

    double sum = stod(st1) + stod(st2);
    string st3 = "Sum is: " + format("{:5g}", sum);
    sumLbl->SetLabel(st3);
}
```

```
// Clear button click event
void Builder::clearClicked(wxCommandEvent& event) {
    num1->SetLabel("");
    num2->SetLabel("");
    sumLbl1->SetLabel("Sum is:");
}

```

Note that when you fetch text from a wxTextCtrl entry field, the method assumes that there may be several lines of text in the window, and you are asking for the first line by GetLineText(0) and converting from a wxString to a string with the ToString() method.

So, in the **addClicked** method, you fetch each entry as a string, and then use the C conversion method **stod** (string to double) to produce a number you can add to another.

And, in the **clearClicked** method, you simply set the contents of the two entry fields and the Sum label to a zero length string. Note that these SetLabel methods will accept a C++ string and automatically promote it to a wxString automatically.

Command Buttons

While for simple programs like this one, it is not uncommon to Bind the click event to the two click event methods above, there is a more general way to handle this by creating a Command Button. In this case, the button itself processes the click event and calls the click event function in the Builder, or wherever else it might reside.

We start by creating a basic DButton abstract class that calls an empty **cmd** method.


```

class DButton : public wxButton {
protected:
    Builder* bld;

public:
    DButton(wxPanel* panel, int id,
            const std::string label,
            Builder* bld, wxApp* app) :
        wxButton(panel, id,
            wxString::wxString(label), wxDefaultPosition,
            wxDefaultSize)
    {
        this->bld = bld;
        Bind(wx.EVT_BUTTON, &DButton::comd, this);
    }

    //abstract method to be completed in derived classes
    virtual void comd(wxCommandEvent& event) =0;
};

```

So, as you can see, the DButton constructor Binds the button event to the **comd** method there in the same class. This comd method is empty and has no code. But note that the method is labelled as **virtual** and set to zero. This means that this method *must* be overridden by methods in derived classes. And that is exactly what we do with these two buttons, derived from DButton.

```

// causes the addition
class AddButton : public DButton {
public:
    AddButton(wxPanel* panel, int id,
              Builder* bld, wxApp* app) :
        DButton(panel, id, string("Add"), bld, app) {}

    void comd(wxCommandEvent& event) {
        bld->addClicked(event);
    }
};

```

Here the add button has **comd** method which calls the **addClicked** method in the Builder class. Note that these derived classes *do not* need to have a Bind call, because it is in the base

DButton class. This is true, of course for the Clear button as well:

```
//clears the entry fields and sum labell
class ClearButton : public DButton {
public:
    ClearButton(wxPanel* panel, int id,
                Builder* bld, wxApp* app) :
        DButton(panel, id, "Clear", bld, app) {}

    void comd(wxCommandEvent& event) {
        bld->clearClicked(event);
    }
};
```

This approach is an example of the Command Design Pattern, where the widget itself calls the function that does the processing. And if you move the **addClicked** and **clearClicked** methods to another class called a Mediator, that is an example of the Mediator Design Pattern, which can handle all the interactions among GUI widgets. We'll see it used in code in following chapters. And of course, the Command pattern can apply to menu clicks, checkbox and Radiobutton clicks and even keystrokes in an entry field, so it is quite general.

Menus

Creating menus in a wxWidgets window is really very simple. The menu system is made up of a wxMenuBar along the top, with wxMenu objects making up the top line entries. You can add as many wxMenuItems under each wxMenu object as you want. A simple example might be one that had two Menu objects, each with one or more menu entries under it.

File	Help
Hello	About
Open	
Clear	

Exit	

The actual program is shown in Figure 13-8:

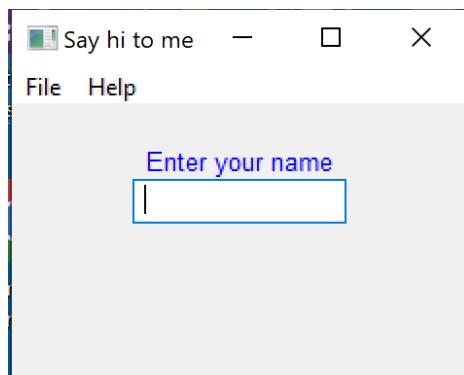


Figure 13-8 - Menu display of HiThere program

While MenuItems are much like Buttons in how they respond to events, programming of them is a little different. Every menu item must have a unique ID. While there are pre-coded Stock Items[5] for most common menu actions, you will need to create your own frequently once you get beyond wxHELP and wxEXIT. There are over 70 predefined stock symbols, but they aren't terribly useful since you can make a set of numbers yourself using the **enum** approach described next.

For this example program, we created a little enumerated list of constants that can be used as MenuItem IDs.

```
enum menuKeys {mkHELLO, mkOPEN, mkHi, mkCLEAR};
```

These keys start at zero unless you specify a specific value for one or more of them.

Using these, we can create the entire menu from the menus and menu items.

```
//Create a File menu on the menu bar
wxMenu* menuFile = new wxMenu;
```

```

menuFile->Append(mkHELLO, "&Hello...\tCtrl+H",
"Help string shown in status bar for this menu item");

menuFile->Append(mkOPEN, "&Open");
menuFile->Append(mkCLEAR, "&Clear");
menuFile->AppendSeparator();
menuFile->Append(wxID_EXIT, "E&xit");

//create a Help menu
wxMenu* menuHelp = new wxMenu;
menuHelp->Append(wxID_ABOUT);

//add the Menus to the MenuBar
wxMenuBar* menuBar = new wxMenuBar;
menuBar->Append(menuFile, "&File");
menuBar->Append(menuHelp, "&Help");

frame->SetMenuBar(menuBar);

```

Note that the Append method actually creates a wxMenuItem directly. You can also do this in two steps, which might be useful if you need to modify the menu item.

```

wxMenuItem* mquit = new wxMenuItem(menuFile,
    wxID_EXIT, wxT("E&xit\tCtrl+X"));
menuFile->Append(mquit);

```

Note that the About menu does not have text provided, because the Stock Item table has “About” as the standard label for wxID_ABOUT. By contrast, we did provide a text label for wxID_EXIT, because the default label in the Stock Items table is “Quit” rather than the expected “Exit.”

Shortcuts and accelerators

For every MenuItem, you can pick an *accelerator character* by preceding it with an ampersand (&) in the label. So, for example to exit from the program you would hold down Alt and then select F and then X.

You can also pick a *shortcut character*, as we do with the Hello menu item. Pressing Ctrl and H together executes the Hello

menu item directly. You can precede any character with Shift, Alt or Ctrl to create these shortcut characters.

Radio or check menuitems

By adding one of the flags `ITEM_CHECK` or `wxITEM_RADIO`, you can turn any menu item into an item with a radio button or checkbox. These are checked or unchecked each time you select them, and you can check them in your code using the `isChecked` method.

Binding MenuItems

Binding `MenuItem`s to action routines is slightly different than for `Buttons` and the like. The `Bind` methods all require that you refer to that menu item's ID. So, the `Bind` methods for this simple demo program are:

```
Bind(wx.EVT_MENU, &Builder::OnClick, this, mkHELLO);
Bind(wx.EVT_MENU, &Builder::Exit, this, wxID_EXIT);
Bind(wx.EVT_MENU, &Builder::Clear, this, mkCLEAR);
Bind(wx.EVT_MENU, &Builder::About, this, wxID_ABOUT);
Bind(wx.EVT_MENU, &Builder::fileOpen, this, mkOPEN);
```

Then you can easily write the simple methods:

```
void Builder::Exit(wxCommandEvent& event) {
    frame->Close(true);
}
void Builder::Clear(wxCommandEvent& event) {
    name->SetLabelText("");
    greeting->SetLabelText("");
}
```

The `OnClick` method is just the same as it was for the `HiThere` program above.

The `About` menu item pops up a simple `Info Dialog`:

```
void Builder::About(wxCommandEvent& event) {
    wxMessageDialog* dlg = new wxMessageDialog(NULL,
        wxT("A simple menu demo"), wxT("Info"), wxOK);
    dlg->ShowModal();
}
```

This is shown in Figure 13-9.

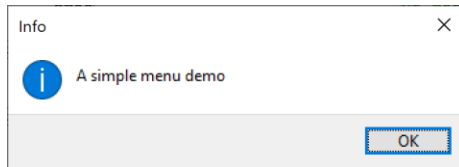


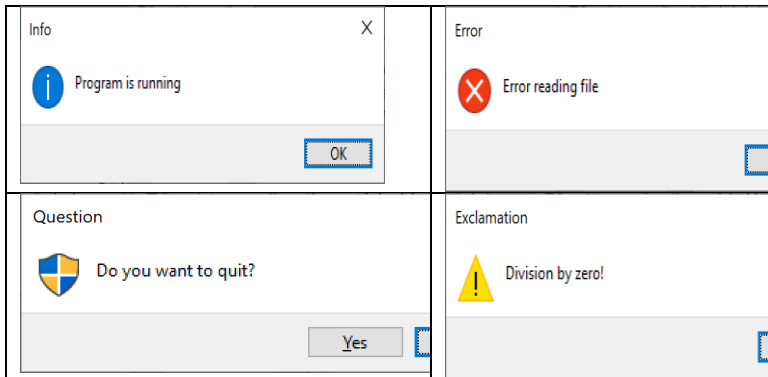
Figure 13-9 - An Info dialog for the About menu item

Dialog Boxes

You can make several useful, common dialog boxes from the **wxMessageDialog** widget varying the icons and buttons you select by ORing these symbols together:

wxOK, wxCANCEL, wxYES_NO, wxHELP, wxNO_DEFAULT, wxCANCEL_DEFAULT, wxYES_DEFAULT, wxYES_DEFAULT, wxOK_DEFAULT, wxICON_NONE, wxICON_EXCLAMATION, wxICON_ERROR, wxICON_HAND, wxICON_QUESTION, wxICON_INFORMATION, wxCENTRE and, wxSTAY_ON_TOP.

Here are four of the most common ones, as suggested by the Zetcode tutorial:



You can create these dialogs with the following calls to the MessageDialog:

```
wxMessageDialog* dlg = new wxMessageDialog(NULL,
```

```

        wxT("Program is running"), wxT("Info"),
        wxOK|wxICON_INFORMATION );
wxMessageDialog* dlg = new wxMessageDialog(NULL,
        wxT("Error reading file"), wxT("Error"),
        wxOK | wxICON_ERROR);

```

Unlike dialogs in other GUI systems, the dialog does not return any value directly. Instead, you must check the value of `dlg.ShowModal()`, which may be `wxID_OK`, `wxID_CANCEL`, `wxID_YES`, `wxID_NO` or `wxID_HELP`, and take appropriate action.

The File Dialog

There are a number of specialized dialogs available for selecting colors and fonts and the like, but one you are most likely to use is the **wxFileDialog** for opening and saving files. It has the form:

```

wxFileDialog
openFileDialog(this, _
    "prompt string",
    "default directory",
    "default file",
    "wildcard filter (*.jpg)",
    flags);

```

Where the flags can be some of the following:

```

wxFD_OPEN or wxFD_SAVE, //is either an open or save
wxFD_OVERWRITE_PROMPT, //prompt to confirm a file
                        //may be overwritten
wxFD_FILE_MUST_EXIST,
wxFD_MULTIPLE,
wxFD_CHANGE_DIR      //change to the current directory

```

Installing wxWidgets

You can download prebuilt binaries for all platforms [8]. For Windows, and Visual Studio, see Reference 9. You can then find Windows solution files for building debugging and release versions for both 64-bit and 32-bit systems.

To create wxWidget programs your project properties should have include file paths set to

```
$(wxwin)\include;$(wxwin)\include\msvc;$(IncludePath)
```

And your library path to

```
$(wxwin)\lib;$(LibraryPath)
```

Where \$wxwin is an environment variable you set to the top level directory where wxWidgets is installed.

Under C/C++ Preprocessor, Preprocessor Definitions should be set to

```
_DEBUG;  
MOREADDING_EXPORTS;  
_CONSOLE;  
_CRT_SECURE_NO_DEPRECATED=1;  
_CRT_NON_CONFORMING_SWPRINTFS=1;  
_SCL_SECURE_NO_WARNINGS=1;  
__WXMSW__;  
_UNICODE;  
_WINDOWS;  
NOPCH;  
Win32_LEAN_AND_MEAN;  
%(PreprocessorDefinitions)
```

Example Programs on GitHub

In cases where there are multiple files as part of a project, they are all stored in a folder together.

- wxHello.py – wxPython version of Hello World
- Frame1.cpp – Hello World title in plain window.
- Frame2.cpp – Simple window with title bar and Greetings label displayed
- Frame3.cpp – Same window using BoxSizer.
- Hithere.cpp – BlueLabel, entry field and “Say hi” button that reads the name and says Hi to it.

- Add2widgets.cpp – Adds 2 numbers and displays answer. Uses GridBagSizer and the Command pattern.
- Moreadding.cpp – same as above only using a Mediator class instead of putting code in the Builder.
- HelloMenu.cpp – Adds a menu to the Hithere.cpp program above.
- Dialogs.cpp – illustrates 4 types of wxMessageDialogs.
- Preproc.txt – The text of the preprocessor definitions.

References

1. <https://docs.wxwidgets.org>
2. Julian Smart and Kevin Hock, *Cross-Platform GUI Programming with wxWidgets*, Prentice-Hall, 2008.
3. Better examples: https://wiki.wxwidgets.org/Main_Page and <https://zetcode.com/gui/wxwidgets/>
4. A complete list of color names can be found at https://docs.wxwidgets.org/3.0/classwx_colour_database.html
5. A complete list of some 70 stock menu names can be found at https://docs.wxwidgets.org/3.0/page_stockitems.html
6. Installing wxWidgets for Visual Studio. <https://www.youtube.com/watch?v=1fZL13jIbFQ>
7. Using wxWidgets in CLion. <https://forums.wxwidgets.org/viewtopic.php?t=45198>
8. Installing wxWidgets. https://docs.wxwidgets.org/3.2.3/overview_install.html
9. Installing wxWidgets for Windows and Visual Studio https://docs.wxwidgets.org/3.2.3/plat_msw_binaries.html

14. Choices and Listboxes

The two most common ways of giving your user choices are radio buttons and checkboxes. Except for the shape of the button (round vs square) and whether you can check more than one, they are remarkably similar to create.

RadioButtons

If you want your user to pick only one of several choices, radio buttons are your best bet. You simply create a panel and insert all the individual buttons in that panel. This makes them all part of the same group, and you can only select one button from a group. If you want another grouping as well, just put them in another panel. The wxWidgets system also allows you to change to a new group in the middle of adding buttons to the same panel by adding the **wxRB_GROUP** modifier to that button, but this is not a great idea, since it could be visually confusing to your user.

So, to create a group of **wxRadioButtons** you simply create a panel and add them, usually using a vertical BoxSizer. We make the following call for each button.

```
cred = new wxRadioButton(panel, wxID_ANY,  
                          wxString("Red"));  
btSizer->Add(cred);
```

The result looks like the left image in Figure 14-1. Clearly this is a little crowded.

A better way is to create a little method that adds space to the left of each button and some space between the buttons. That code is simply

```

wxRadioButton* Builder::addButton( wxPanel* pnl, string
label) {
    //create the button
    wxRadioButton* cbut =
        new wxRadioButton(pnl, wxID_ANY, wxString(label));
    // put it in a horizontal BoxSizer space to the left
    wxBoxSizer* hbox = new wxBoxSizer(wxHORIZONTAL);
    hbox->AddSpacer(20);
    hbox->Add(cbut);
    btSizer->Add(hbox);
    btSizer->AddSpacer(10); //add space after button
    return cbut;
}

```

Then in the main builder routine we call this function 3 times:

```

cred = addButton(panel, "Red");
cblue = addButton(panel, "Blue");
cgreen = addButton(panel, "Green");

```

The result is on the right side of Figure 14-1.

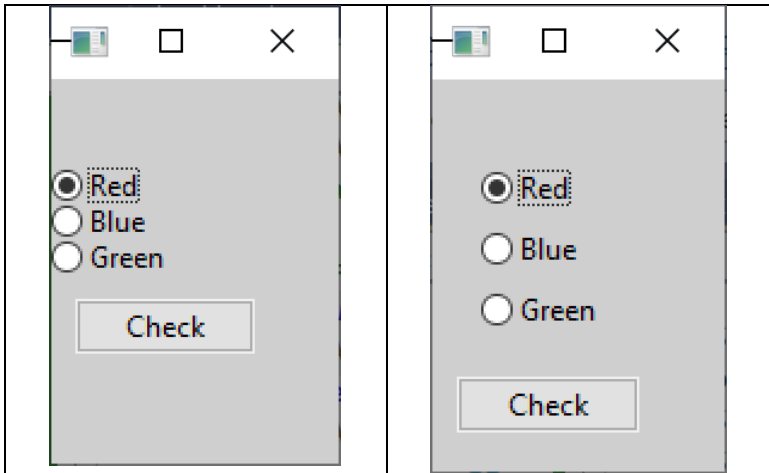


Figure 14-1 -- Radio buttons without and with spacers

Reading the Radio buttons

Finding out which button is selected is much simpler than in Python's tkinter approach. You can iterate through the buttons

and find which one is selected and place that label at the top of the window using the **GetFirstInGroup()** and **GetNextInGroup()** methods. Then for each button, you call **GetValue()**. If it returns **true**, that button is selected.

```
void Builder::OnClick(wxCommandEvent& event) {
    //get first button
    wxRadioButton* cbut = cred->GetFirstInGroup();
    do {
        if (cbut->GetValue()) //true if selected
            topLabel->SetLabelText(cbut->GetLabelText());
        cbut = cbut->GetNextInGroup(); //get next button
    } while (cbut != NULL); //until done
}
```

The result is show in Figure 14-2.



Figure 14-2 -- Label contains the text of the selected button

Responding to RadioButton clicks

Let's consider a program where the display changes whenever you click on any of the radio buttons. This means that instead of using that Check button to check the settings of the radio buttons you see a result as soon as you click on one. Figure 14-3 shows a simple example program that changes the color of the right-hand panel as soon as you click on a button.

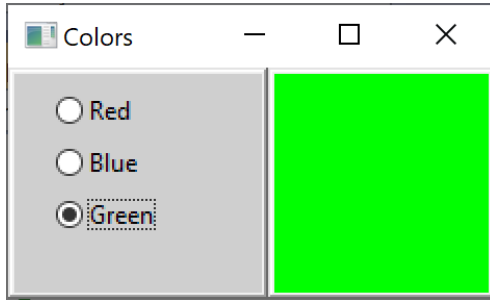


Figure 14-3 --Change colors on click

The GUI is made of two panels, with the left panel the same three-color buttons. The right panel's background color changes whenever you click on one of them.

The simplest way to do this is to bind each of the three buttons to a click event:

```
cred = addButton(leftPanel, "Red");
cred->Bind(wxEVT_RADIOBUTTON, &Builder::redClick, this);

cblue = addButton(leftPanel, "Blue");
cblue->Bind(wxEVT_RADIOBUTTON, &Builder::blueClick, this);

cgreen = addButton(leftPanel, "Green");
cgreen->Bind(wxEVT_RADIOBUTTON, &Builder::greenClick,
            this);
```

Then the click events simply change the right panel's color. Note that you have to refresh the panel for that color to change:

```
void Builder::redClick(wxCommandEvent& event) {
    rightPanel->SetBackgroundColour("red");
    rightPanel->Refresh();
}
void Builder::blueClick(wxCommandEvent& event) {
    rightPanel->SetBackgroundColour("blue");
    rightPanel->Refresh();
}
void Builder::greenClick(wxCommandEvent& event) {
    rightPanel->SetBackgroundColour("green");
    rightPanel->Refresh();
}
```

This will work for a small number of buttons, but for a larger number it can get a bit unwieldy.

A second way to select the right action is to bind all of the buttons to a single **onClick** method and then iterate through the buttons to find the selected one, much as we did earlier.

```
void Builder::onClick(wxCommandEvent& event) {
    //get first button
    wxRadioButton* cbut = cred->GetFirstInGroup();
    do {
        if (cbut->GetValue()) {
            wxString lbl = cbut->GetLabelText();
            lbl.LowerCase();
            rightPanel->SetBackgroundColour(wxColor(lbl));
        }
        cbut = cbut->GetNextInGroup(); //get next button
    } while (cbut != NULL);           //until done
    rightPanel->Refresh();
}
```

Finding the calling object

The third way to handle this is to find the object that caused the event. Here we get that object from the event using the **GetEventObject** method. Then we cast it to the **wxRadioButton*** type and fetch the label, representing the color.

```
void Builder::onClick3(wxCommandEvent& event) {
    //get the object that caused the event
    auto winobj = event.GetEventObject();
    //cast to wxRadioButton*
    wxRadioButton* cbut = (wxRadioButton*)winobj;
    wxString lbl = cbut->GetLabelText();
    lbl.LowerCase();
    rightPanel->SetBackgroundColour(wxColor(lbl));
    rightPanel->Refresh();
}
```

ListBoxes

ListBoxes in wxWidgets are pretty easy to use. In fact, they are simpler than the ones in tkinter. You can create a simple ListBox:

```
lbox = new wxListBox(panel, wxID_ANY,
                    wxDefaultPosition, wxSize(150, 100),
                    0, NULL, wxLB_SINGLE);
```

Instead of a specific size, you could put in **wxDefaultSize**, but the default sizes are often pretty big. There are a number of styles you can choose from for a listbox, and some of them can be ORed together where this makes sense.

- `wxLB_SINGLE` (equals 0) – can select a single entry
- `wxLB_MULTIPLE` – can select more than one entry
- `wxLB_EXTENDED` - can select more using the Shift and Ctrl keys
- `wxLB_HSCROLL` – create horizontal scroll bar for long entries (Windows only)
- `wxLB_ALWAYS_SB` – always show a vertical scroll bar
- `wxLB_NEEDED_SB` – create a vertical scroll bar if needed (default)
- `wxLB_NO_SB` – never create a vertical scrollbar (Windows and GTK only)
- `wxLB_SORT` – sort the list

If you don't need any of those options, you can leave out the last three arguments:

```
lbox = new wxListBox(panel, wxID_ANY,
                    wxDefaultPosition, wxSize(150, 100));
```

You can also load some entries into the listbox in the constructor:

```
wxString choices[2] = { "Anne", "Betty" };
lbox = new wxListBox(panel, wxID_ANY,
                    wxDefaultPosition, wxSize(150, 100), 2, choices,
                    wxLB_MULTIPLE);
```


As shown, you must indicate the number of strings the listbox is to load (here, 2) along with the array of strings.

The other way to add strings to the listbox is using the **Append** method:

```
lbox->Append(wxT("Fred"));
lbox->Append(wxT("Sally"));
lbox->Append(wxT("Sam"));
lbox->Append(wxT("Bridget"));
```

Together, the two of these produce the listbox in Figure 14-4.

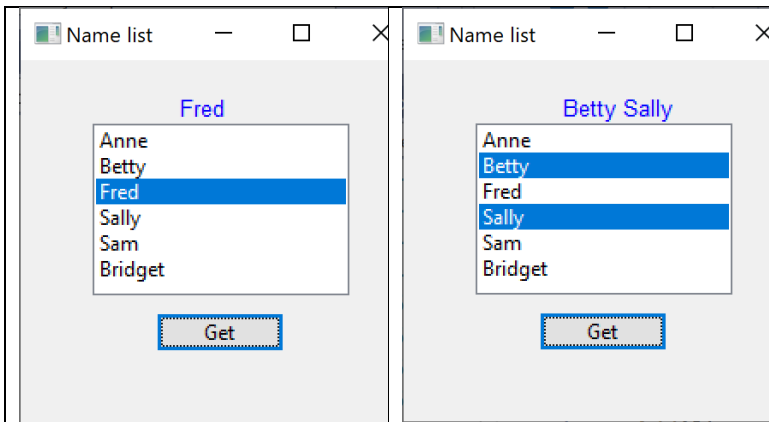


Figure 14-4 –Single(a) and multiselect(b) listboxes

To get the line or lines selected in the ListBox, you have two choices. For a single selection listbox, you get the index of the line selected, and then fetch the string at the position if it is a positive number. If it is negative, no line has been selected:

```
// get the index and fetch that line
int index = lbox->GetSelection();
if (index >=0)
    title->SetLabel(lbox->GetString(index));
```

If you try to fetch a string with a negative index, an error will occur.

The second approach is for multi-select listboxes, but in fact works for single select just as well. You create a **wxArrayInt**

object and pass it to the **GetSelections** method. It returns the number of lines selected and the indexes of those selections in the `ArrayInt` object. You fetch each one using the **Item** method of that array:

```
//This will work with the multiple and the single versions
wxArrayInt selections; //create the empty array
int count = lbox->GetSelections(selections); //load it
string text = ""; //append all selections here
for (int i = 0; i < count; i++) {
    int index = selections.Item(i);
    text += lbox->GetString(index) + " ";
}
title->SetLabel(text); // set all the text in label
```

The resulting display is shown in Figure 14-4b.

It is important to note that you cannot use the **GetSelection** method in a multi-select listbox. It will cause an error.

CheckListBoxes

You can use the same code to display a listbox with checks by calling `wxCheckListBox` instead of the `wxListBox`. The only difference is for each element, you have an **isChecked** method you can call. Checks are separate from whether a line is selected or not. This is illustrated in Figure 14-5.

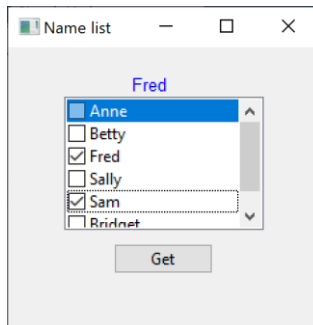


Figure 14-5 -- Example of a `CheckListBox`

The StateLister Application

Now that we've spent some time on listboxes, let's consider a more elaborate case, where we have an array of State classes that we create by reading in the states.txt file. Each state has a name, abbreviation, capital and founding date. The app looks like that in Figure 14-6.

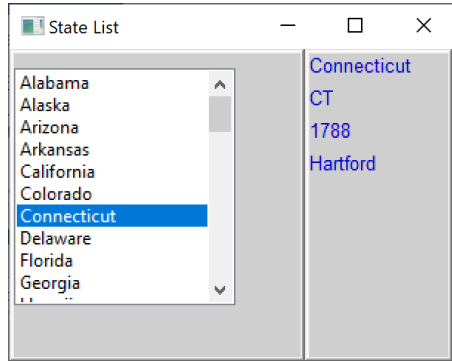


Figure 14-6 -The StateLister app

The State class is pretty obvious. It stores those four properties for each state and lets you fetch them from each instance using getter methods. The constructor for each State instance uses our StrFuncs utility class to split up the comma-separated list and put the four values into the instance variables:

```
State::State(std::string line) {
    vector<std::string> tokens =
        StrFuncs::split(line, ",");
    name = tokens[0];
    abbrev = tokens[1];
    date = tokens[2];
    capital = tokens[3];
}
string State::getName() { return name; }
string State::getAbbrev() { return abbrev; }
string State::getDate() { return date; }
string State::getCapital() { return capital; }
```

And the surrounding **StateList** class reads each line, creates a State instance and stores it in a vector where you can easily retrieve it.

```
//reads and keeps the State list
class StateList {
private:
    vector<State*> states;

public:
//read in the States file and store each State in a vector
    StateList(string fileName) {

        ifstream myfile(fileName);
        string line;
        if (myfile.is_open()) {
            while (getline(myfile, line)) {
                states.push_back(
                    new State(line));
            }
            myfile.close();
        }
        else cout << "Unable to open file";
    }
//get the whole vector
    vector<State*> getStates() {
        return states;
    }
//get a single state
    State* getState(int index) {
        return states[index];
    }
};
```

Using a Mediator Class

This program consists of a number of classes and widgets, and it is probably time to consider using a Mediator class to handle the interactions between them. The Mediator knows about the StateList and State classes as well as the listbox and the four

labels in the right panel. So when there is a click on the listbox, the `OnClick` event just tells the Mediator to handle it.

```
void MyFrame::onClick(wxCommandEvent& ev) {
    med->listClick(ev);
}
```

Then, the Mediator fetches the correct State from the `StateList` class and loads the labels:

```
void Mediator::listClick(wxCommandEvent& ev) {
    //first get the entry clicked on
    int index = stateListBox->GetSelection();

    //then get the state object at that index
    State* state = stateList->getState(index);

    //load the labels with that State's data
    lbName->SetLabelText(state->getName());
    lbAbbrev->SetLabelText(state->getAbbrev());
    lbDate->SetLabelText(state->getDate());
    lbCapital->SetLabelText(state->getCapital());
}
```

The only real overhead in using a Mediator is passing the variables it needs into the Mediator class. Some of this happens in the constructor:

```
med = new Mediator(stateList);
```

and the rest in a couple of **set** methods:

```
//read in states
StateList* sList = new StateList("States.txt");
med->setStateList(sList);
med->setLabels(lbName, lbAbbrev, lbDate,
             lbCapital);
```

And the Mediator loads the listbox from the `StateList` vector:

```
void Mediator::setStateList(StateList* slist) {
    stateList = slist;

    //get the vector and load the listbox
    vector<State*> states = stateList->getStates();
```

```

for (int i = 0; i < states.size(); i++) {
    stateListBox->Append(
        wxString(states[i]->getName()));
}
}

```

This takes the complexity of the loading and clicking out of the Builder, which is only supposed to create the GUI and puts all the interactions in a single place: the Mediator. As your GUI programs become more complex this Mediator Design Pattern is an ideal way to group your GUI and other object interactions.

The ComboBox

The combo box (Figure 14-7) is pretty much the same as a regular list, except that it drops down instead of taking up a lot of space all the time:

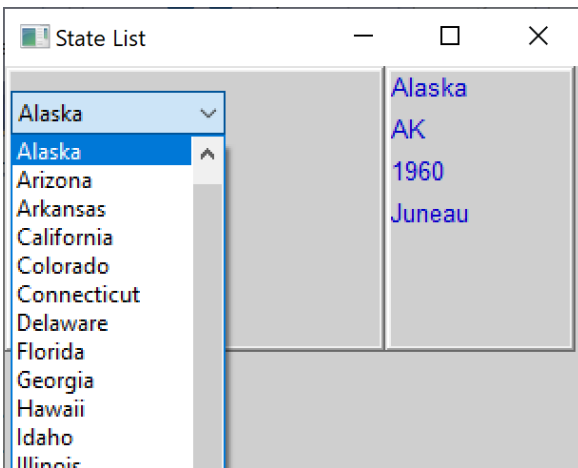


Figure 14-7 -- The state list using a combo box

Loading the combo box is just the same as for a regular listbox:

```
// create a vector list of State objects
// and then insert the names of the states in the combobox
std::vector<State*> states = slist->getStates();
for (int i = 0; i < states.size(); i++) {
    stateList->Append(
        wxString(states[i]->getName()));
}
}
```

The only other difference is the name of the click event:

```
//connect item click to the onClick method
Bind(wx.EVT_COMBOBOX, &MyFrame::onClick, this);
```

Checkboxes

The **wxCheckBox** is very similar to the **wxRadioButton**, except that checkboxes are not grouped and you can select as many boxes as you want. Since they are not grouped, there is no convenient way to iterate through them to see what has been checked. Therefore, most people keep a vector with pointers to the checkboxes so you can quickly find out which are checked.

In this pizza ordering example (Figure 14-8), we click on the checkboxes and the ordered toppings appear in the righthand listbox.

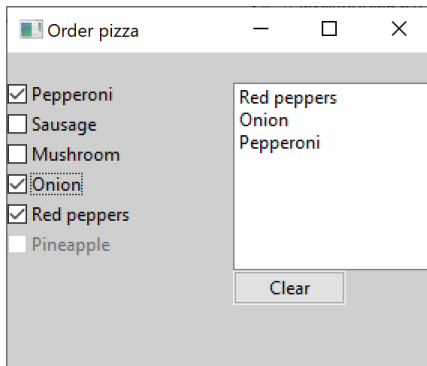


Figure 14-8 --Pizza ordering using check boxes.

Note that there is no “Get” or “Order” button. Clicking on any checkbox immediately adds that topping to the list. There are a couple of ways you can do this. One is to Bind each checkbox to the OnClick event handler, but a simpler way is to simply bind the left panel to the checkbox click event, since the events propagate up into the container that hold them.

We create the checkboxes in the code below.

```
leftSizer = new wxBoxSizer(wxVERTICAL);
leftPanel->SetSizer(leftSizer);
leftSizer->AddSpacer(20);
addCheckBox("Pepperoni", leftSizer);
addCheckBox("Sausage", leftSizer);
addCheckBox("Mushroom", leftSizer);
addCheckBox("Onion", leftSizer);
addCheckBox("Red peppers", leftSizer);
wxCheckBox* pine =
    addCheckBox("Pineapple", leftSizer);
pine->Disable();
Bind(wxEVT_COMMAND_CHECKBOX_CLICKED,
     &Builder::OnClick, this);
```

Note that in honor of the internet joke that “pineapple does not belong on pizza,” we disable that choice. The addCheckBox method below adds the checkbox to the sizer *and* to the vector where you can check out its contents later.

```
//Adds a checkbox to the left panel's sizer
wxCheckBox* Builder::
addCheckBox(string label, wxSizer* leftChecks) {
    wxCheckBox* cb1 = new wxCheckBox(leftPanel,
                                     wxID_ANY, wxString(label));
    leftChecks->Add(cb1);
    leftChecks->AddSpacer(5);
    checks.push_back(cb1);    //and to the vector list
    return cb1;
};
```

Finally, the OnClick event clears the listbox and refills it with the currently check items:

```
//fills list with currently checked items
void Builder::OnClick(wxCommandEvent& event) {
```



```

wxCheckBox* cb1;
orderList->Clear();      //clear the listbox

//and re-fill it from the checkboxes
for (int i = 0; i < checks.size(); i++) {
    cb1 = checks.at(i);
    if (cb1->IsChecked()) {
        wxString label = cb1->GetLabel();
        orderList->InsertItems(1, &label, 0);
    }
}
}

```

Checkbox styles

You can change the checkbox styles with these flags, some of which can be ORed together:

- wxCHK_2STATE – creates standard 2 state checkbox
- wxCHK_3STATE – creates 3 state checkbox
- wxCHK_ALLOW_3RD_STATE_FOR_USER – user can click all 3 states
- wxALIGN_RIGHT – puts label to left of checkbox

Figure 14-9 shows all three states and both alignments.

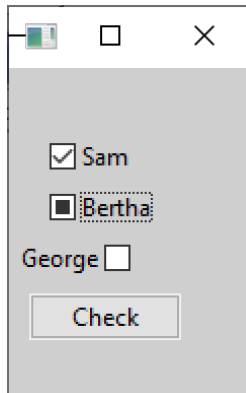


Figure 14-9 --All 3 states of checkbox, and both alignments

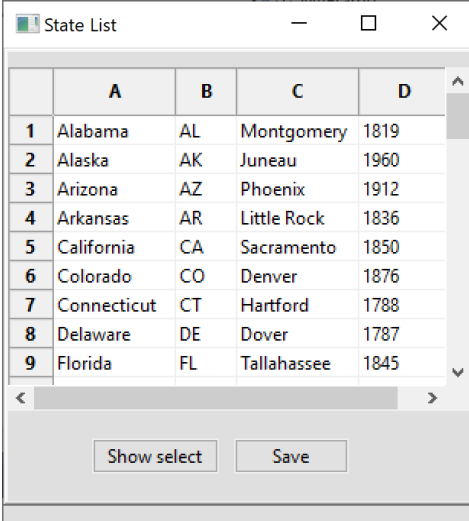
To get all 3 states, you must OR the second the third flags together and include the two default placeholders before specifying the flags.

```
wxCheckBox* cbut =
    new wxCheckBox(pnl, wxID_ANY, wxString(label),
                  wxDefaultPosition, wxDefaultSize,
                  wxCHK_3STATE|wxCHK_ALLOW_3RD_STATE_FOR_USER);
```

For a 3-state checkbox, there are 3 values, `wxCHK_UNCHECKED`, `wxCHK_CHECKED`, and `wxCHK_UNDETERMINED`. You can access these with the **Get3StateValue** and **Set3StateValue** methods, and you can change the state of the checkbox by using the **Set3StateValue** method to change the state of the checkbox's operation.

Displaying tables in a grid

The **wxGrid** widget is a really flexible, easy-to-use table display that can even let you edit the cells in real time. In Figure 14-10 we see how it looks in its simplest form:



	A	B	C	D
1	Alabama	AL	Montgomery	1819
2	Alaska	AK	Juneau	1960
3	Arizona	AZ	Phoenix	1912
4	Arkansas	AR	Little Rock	1836
5	California	CA	Sacramento	1850
6	Colorado	CO	Denver	1876
7	Connecticut	CT	Hartford	1788
8	Delaware	DE	Dover	1787
9	Florida	FL	Tallahassee	1845

Figure 14-10 -- a `wxGrid` display of the states

Creating this grid amounts to specifying the grid dimensions and setting the column widths:

```
//create the grid, 50 x 4
stateGrid = new wxGrid(leftPanel, wxID_ANY,
    wxDefaultPosition, wxSize(400, 225));
stateGrid->CreateGrid(50, 4);
stateGrid->SetColSize(1, 40);//set column sizes
stateGrid->SetColSize(3, 60);
stateGrid->SetRowLabelSize(30);
pnlSizr->Add(stateGrid);
```

Then to load it, we use the familiar **States** class to read in the states.txt file and then get the states one at a time and load each row:

```
// and then insert the values for the states into the grid
vector<State*> states = slist->getStates();
for (int i = 0; i < states.size(); i++) {
    stateGrid->SetCellValue(i, 0,
        wxString(states[i]->getName()));
    stateGrid->SetCellValue(i, 1,
        wxString(states[i]->getAbbrev()));
    stateGrid->SetCellValue(i, 2,
        wxString(states[i]->getCapital()));
    stateGrid->SetCellValue(i, 3,
        wxString(states[i]->getDate()));
}
```

By default, the cells are editable unless you call the grid's **DisableCellEditControl()** method.

To save the results of any cell edit, you need to catch that event by Binding it to some code to save the new value.

```
//connect item click to the onClick method
Bind(wxEVT_GRID_CELL_CHANGED,
    &MyFrame::saveClick, this);
```

Then, to save that edit you need to fetch that string and put it back into the state array:

```
//Save data for selected state
void Mediator::saveState(wxGridEvent& gev) {
    int row = gev.GetRow();
```

```

int col = gev.GetCol();
//get the changed cell text
string gtext = stateGrid->GetCellValue(row,
    col).ToStdString();
State* state = stateList->getState(row);

//convert column number to a property
//and save it back into the state vector
switch (col) {
    case 0: state->setName(gtext); break;
    case 1: state->setAbbrev(gtext); break;
    case 2: state->setCapital(gtext); break;
    case 3: state->setDate(gtext); break;
}
}

```

Now, that data are only in memory, and to save it in a file, you must click on that **Save** button. It will run through the entire vector of states, convert each to a comma-separated list and store it in a new file, in this case **states1.txt**.

```

void Mediator::saveStates() {
    //save all the states to a new file
    ofstream stFile("states1.txt");
    vector<State*> stateVector =
        stateList->getStates();
    for (auto iter(stateVector.begin());
        iter != stateVector.end(); ++iter) {
        State* st = *iter;

        //get comma sep string
        string output = st->getLineString();
        stFile << output << endl;    //and write it
    }
    stFile.close();
}

```

Selecting Grid regions

The wxGrid widget lets you select any region of cells and do with it what you will. In we show a selected region, and the popup window of the data in those cells:

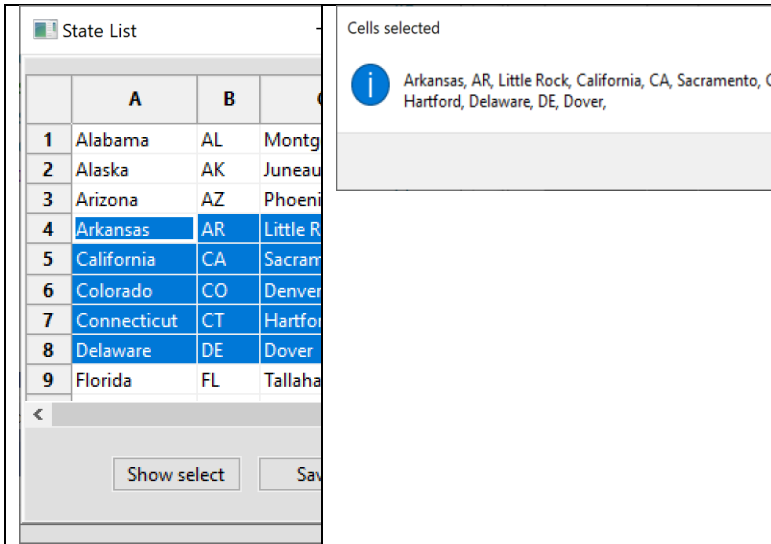


Figure 14-11 --wxGrid region selection and popup window showing that data

```
void Mediator::rangeClick(wxCommandEvent& ev) {
    //strings saved here
    std::vector< wxString> tokens;

    //go through all the rows
    for (int i = 0;
        i < stateGrid->GetNumberRows(); ++i) {
        if (stateGrid->IsInSelection(i, 0)) {
            //go through each column in that row
            for (int c = 0;
                c < stateGrid->GetNumberCols();
                c++) {
                if (stateGrid->IsInSelection(i, c)) {
                    tokens.push_back
                        (stateGrid->GetCellValue(i, c));
                }
            }
        }
    }

    //now create a Message Dialog with the result
    string message = "";
    for (int i = 0; i < tokens.size(); i++) {
        message += tokens[i] + ", ";
    }
    wxMessageDialog* dial = new wxMessageDialog(NULL,
```

```

wxString(message), wxT("Cells selected"), wxOK);
dial->ShowModal();
}

```

Other wxGrid features

You can specify any cell as being a number cell with **SetColFormatNumber**, **SetColFormatFloat**, or **SetColFormatBool**.

The Tree widget

The **wxTreeCtrl** represents a tree with the root at the top and branches of data under it. Creating the tree visual is extremely easy: the more involved part might be representing the data behind it. Skipping, that we'll just create a tree (Figure 14-12) using a couple of state's data:

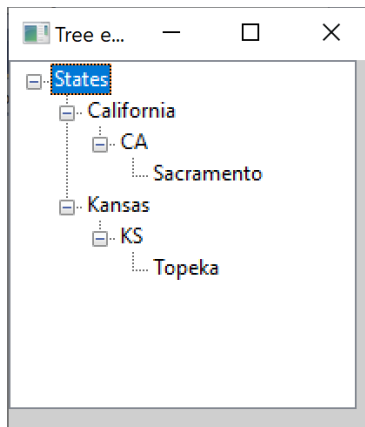


Figure 14-12 --The Tree control with two states shown

The code amounts to creating a root and adding children to it:

```

panel = new wxPanel(frame, wxID_ANY);
wxBoxSizer* sizer = new wxBoxSizer(wxVERTICAL);
panel->SetSizer(sizer);
tree = new wxTreeCtrl(panel, wxID_ANY,
wxDefaultPosition, wxSize(200, 200),
wxTR_DEFAULT_STYLE, wxDefaultValidator,
wxString("state tree"));

```

```

sizer->Add(tree);

wxTreeItemId rootId = tree->AddRoot("States");
wxTreeItemId child =
    tree->AppendItem(rootId, "California");
wxTreeItemId child1 =
    tree->AppendItem(child, "CA");
tree->AppendItem(child1, "Sacramento");

wxTreeItemId newRoot =
    tree->AppendItem(rootId, "Kansas");
wxTreeItemId child2 =
    tree->AppendItem(newRoot, "KS");
tree->AppendItem(child2, "Topeka");
frame->Show();

```

As you can see, the Tree is extremely easy to use. While it isn't the default like in the Grid, it is possible to create a tree with editable labels and edit events by using the `wxTR_EDIT_LABELS` style when you create the tree widget.

Moving on

We've shown you how to create and use all the common widgets in wxWidgets. Nearly all of them are easier to use than the analogous controls in tkinter. In the following chapters we'll look at several other common tools that are analogous to ones available on Python.

Example programs on GitHub

- Radiobuts.cpp – radio buttons with and without spacers
- Radioboxing.cpp – illustrates RadioBoxes.
- RadioColor.cpp – uses 3 onClick events
- RadioColor2.cpp – shows both scanning the list and getting the button from the event
- SimpleListbox.cpp – single and multiselect and check listboxes.
- StateLister – List box of state objects showing details
- StateListCombo – same as State Lister but using a combo box

- PizzaChecker – Adds pizza orders from checkboxes to a listbox
- Check3state.cpp – shows all 3 states of checkbox
- StateListGrid – shows the basic wxGrid display of states
- Treectrl.cpp – a simple Tree control widget example

Part II- Application Development

In this section we discuss libraries for mathematical computations, plotting and connecting to databases.

For math computations we explain how to use the public domain Armadillo library, which provides much the computational features that you find in Python's Numpy library, and which is similar to Matlab in capabilities.

For plotting, we discuss both SciPlot and ROOT, which each have advantages.

And finally, we discuss connecting to databases, with examples connecting to SQLite and to MySQL. We end up developing a framework which will work with either, even though the underlying interface code is significantly different between the two systems.

15. The Armadillo Math Library

Armadillo is a C++ open-source linear algebra library that allows you to manipulate matrices and carry out quite a number of useful computations. It was developed by Conrad Sanderson and Ryan Curtin at the University of Queensland and Griffith University. The library is licensed under the relatively permissive Apache license and the syntax of the library classes are deliberately similar to those in Matlab, and much the same as the classes in Numpy.

Much of Armadillo is built on the OpenBLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package) libraries, here wrapped into useful classes. The library is available for Windows, Macs and Linux. For experimentation and initial development, Windows may be easier, but the math libraries run faster on Linux platforms. It also uses delayed evaluation at compile time to increase computational efficiency. Nearly all of the classes in Armadillo are implemented as templates, and for that reason, you probably should avoid using the C++ **auto** keyword.

Overview of Armadillo Classes

Armadillo contains hundreds of useful classes and functions, briefly summarized here. The complete list is in the online documentation.

- Matrix, Vector, Cube and Field classes
 - Matrices, columns, rows, cubes and fields
 - Determinant, sum, diagonal, transpose, fill with random numbers, etc.
 - Eigen decomposition, inverse, etc.
- Signal and Image Processing
 - Convolution
 - 1D and 2D FFT
 - Interpolation

- Polynomial fitting
- Statistics and Clustering
 - Mean, covariance, correlation
 - Principal component analysis
 - Probability density function

Matrices

The `Mat` class is the fundamental dense matrix object in Armadillo. They are stored column by column. You can create matrices of any of the common types: double, float, complex double, complex float, short, int long and unsigned. Here's a simple matrix creation command:

```
//constructing a matrix
  Mat<double> A (3,4);
```

However, Armadillo defines some simple matrix type names to simplify your programming:

```
mat = Mat<double>
dmat = Mat<double>
fmat = Mat<float>
cx_mat = Mat<cx_double>
cx_dmat = Mat<cx_double>
cx_fmat = Mat<cx_float>
umat = Mat<uword>
imat = Mat<sword>
```

So, you can write more simply:

```
mat B(3, 4);
```

Matrices are created with all elements set to zero, but you can create them using several other patterns:

```
fill::zeros  set all elements to 0
```

```
fill::ones   set all elements to 1
```

<code>fill::eye</code>	set the elements on the main diagonal to 1 and off-diagonal elements to 0
<code>fill::randu</code>	set all elements to random values from a uniform distribution in the [0,1] interval
<code>fill::randn</code>	set all elements to random values from a normal/Gaussian distribution with zero mean and unit variance
<code>fill::value (scalar)</code>	set all elements to specified scalar (Armadillo 10.6 and later)
<code>fill::none</code>	do not initialize the elements

Thus, this statement fills the matrix with random numbers between 0 and 1:

```
mat D(5, 5, fill::randu); //uniform random distribution
D.print();                //print out matrix
```

giving you a 5x5 matrix of random numbers:

```
0.8634 0.8296 0.1600 0.2330 0.6979
0.8899 0.1792 0.7420 0.7058 0.3701
0.3604 0.2807 0.7047 0.7213 0.1271
0.4156 0.2088 0.1175 0.3593 0.2863
0.9300 0.8655 0.7502 0.7527 0.2077
```

You can also perform the standard matrix algebra (add, subtract, multiply, divide) in Armadillo:

```
mat E = D + D;
```

The usual rules apply: you can only add and subtract matrices of the same dimensions, and to multiply or divide, the number rows and columns in one must be the same as the number of columns and rows in the other.

Columns and Rows

Each matrix column is essentially a one-dimensional matrix, and since it is derived from the `Mat` class, most of the methods apply to both columns and rows. Columns and rows are essentially vectors, so they are also referred to by the names **colvec** and **rowvec**. Like matrices, the columns and rows have typedefs for each the common data types: `colvec`, `dcolvec`, `fcolvec`, `cx_colvec` and so forth. They are also defined as `vec`, `dvec`, `fvec`, `cx_vec` and so forth.

You can, of course get any single column or row using the **.row** and **.col** matrix methods:

```
//extract a column and print it
    colvec q = D.col(0);
    q.print();
```

Matrix methods

But the great power of Armadillo lies in the methods you can use on any matrix. There are many functions and operations in the complete list in the documentation, but they include:

Matrix functions

Accu	Sum matrix elements
Affmul	Affine matrix multiplication
Conj	Complex conjugate of each element
Cross	Cross product of 2 matrices
Diagmat	Generate diagonal matrix
Kron	Kronecker tensor product
Norm	Normalize vectors
Rank	Rank of matrix
Trace	Sum of diagonal elements
Trans	Transpose of matrix. Also can use <code>.t()</code> method.
Vectorise	Flatten matrix into vector. Note spelling.

Decompositions, Inverses and Equation Solvers

Chol	Cholesky decomposition
Eig_sym	Eigen decomposition of dense symmetric matrix
Hess	Upper Hessenberg decomposition
Inv	Inverse of square matrix
Svd	Singular value decomposition
Syl	Sylvester equation solver

Signal and Image Processing

Conv	1 D convolution
Fft	1D fast Fourier transform and inverse
Fft2	2D FFT and inverse
Polyfit	Fit a polynomial to a set of points
Polyval	Evaluate polynomial

Matrix transpose

Let's suppose we want to rotate a matrix by 90 degrees. The **trans** method will do this for you. In the demo program below, we create a random number-filled matrix when we click on the Load button, and transpose it when we click on the Trans buttons:

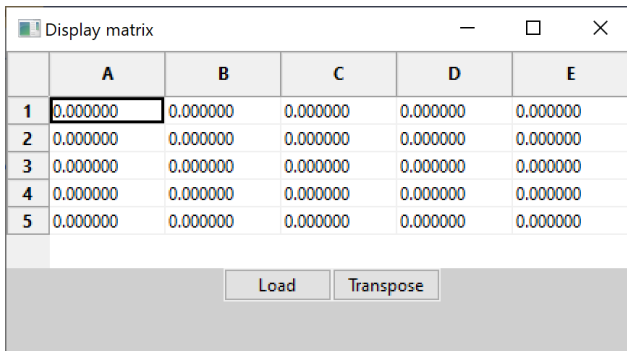


Figure 15-1 - 5 x 5 matrix filled with zeroes

So, as before, the code to create the zero matrix is quite simple. You can access matrices with zero-based indexing just as if they were C++ arrays:

```
//create the matrix
void Mediator::loadClicked() {
    //create a 5x5 matrix
    A.set_size(5, 5);
    A.fill(fill::randu);
    loadGrid(A);
}
```

We load the grid by moving the elements one by one on the usual indexing code. Note that when you access values in a matrix, the indices are enclosed in parentheses (`C(i, j)`), not brackets, since these are arguments to internal methods in the `Mat` object:

```
// and copy it into the wxWidgets grid:
void Mediator::loadGrid(mat C) {
    for (int i = 0; i < C.n_rows; i++) {
        for (int j = 0; j < C.n_cols; j++) {
            numGrid->SetCellValue(i, j, to_string(C(i,
j)));
        }
    }
}
```

The transpose

You could write:

```
mat B = trans(A);
```

or, you can use the more compact method built into the `Mat` class:

```
mat B = A.t();
```

It is this latter method we use in the demo program:

```
void Mediator::transposeClicked() {
```

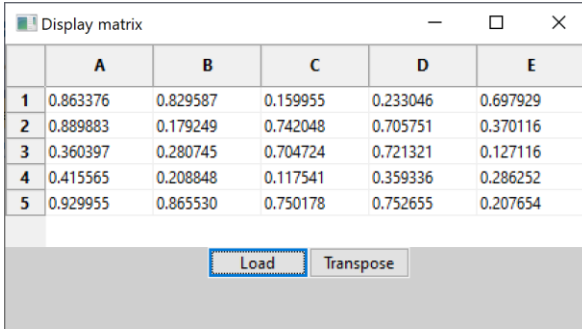


```

mat B = A.t(); //compute transpose
loadGrid(B);  //and load it
}

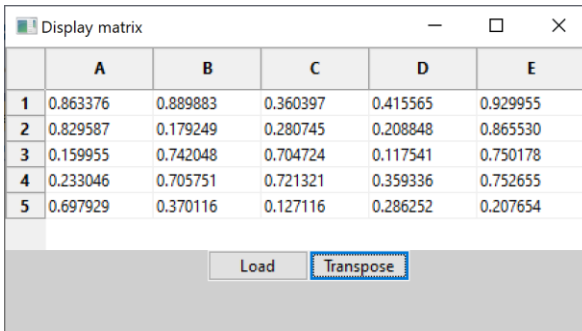
```

The results are show in Figure 15-2 and Figure 15-3.



	A	B	C	D	E
1	0.863376	0.829587	0.159955	0.233046	0.697929
2	0.889883	0.179249	0.742048	0.705751	0.370116
3	0.360397	0.280745	0.704724	0.721321	0.127116
4	0.415565	0.208848	0.117541	0.359336	0.286252
5	0.929955	0.865530	0.750178	0.752655	0.207654

Figure 15-2 - matrix filled with random numbers.



	A	B	C	D	E
1	0.863376	0.889883	0.360397	0.415565	0.929955
2	0.829587	0.179249	0.280745	0.208848	0.865530
3	0.159955	0.742048	0.704724	0.117541	0.750178
4	0.233046	0.705751	0.721321	0.359336	0.752655
5	0.697929	0.370116	0.127116	0.286252	0.207654

Figure 15-3 Matrix transposed

The Fast Fourier transform

You use the Fourier transform` to convert from the time domain to the frequency domain. For example, a plot of a sine wave is a plot of signal versus time. The FFT can quickly convert to signal versus frequency, where there will be a single peak for each frequency in the data.

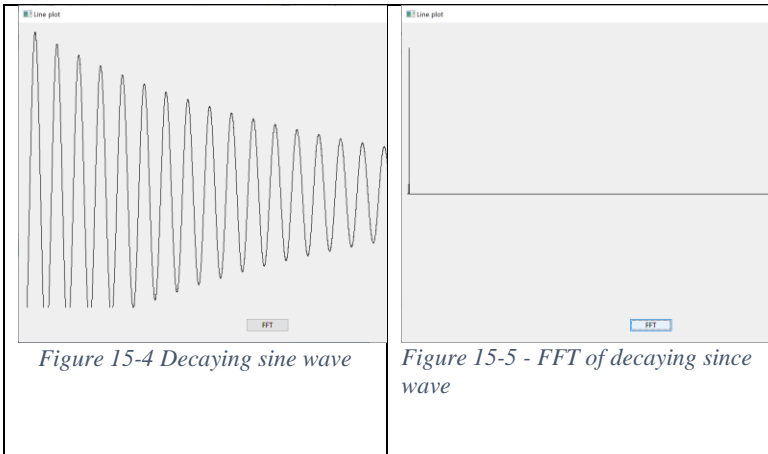
In Figure 15-4, we show a decaying sine wave, representing a single frequency. We added the exponential decay, because the resulting FFT peak will then be a bit wider and easier to see in

Figure 15-5. As you can see, there is one peak representing the single frequency on a plot of signal versus frequency.

The code for calculation the FFT is just as simple as you might think:

```
vec sinedata = vec(waveData); //convert array of floats
cx_vec spec = fft(sinedata); //complex result
```

You start with an array of floats representing the sine data and convert it to a **vec** or **colvec**, which means the same thing in Armadillo. Then you call the **fft** function, which returns a vector of complex numbers. Usually, the input array will have a dimension of a power of 2 for the FFT to work most efficiently. In the example shown below, we generated a decaying sine wave having 16,384 points. The result of the FFT is an array (vector) of 8192 complex numbers, each having a real and imaginary part. In plotting the result, we simply create an array from the real parts of each complex number as shown in Figure 15-5. So, the result is an 8,192 point real array.



The code to create that array is simply:

```

size_t peakCount = spec.size()/2;
peakPoints = new wxPoint[peakCount];
double yval = 0;
for (size_t x = 0; x < peakCount; x++) {
    yval = spec[x].real()/50;
    int y = Height - int(yval/2)- 300; // wave height
    int xcoord = int(x * (float(Width) /
        float(peakCount))) + 10; // x
    wxPoint pt(xcoord, y); //display array
    peakPoints[x] = pt;
}

Refresh();

```

The scaling constants 50, 2 and 300 were determined empirically. In the following chapter, we'll show how to compute them generally.

Curve fitting

One very nice feature buried in Armadillo is polynomial curve fitting. It will generate a polynomial of any order to fit a set of x-y data. For example, Figure 15-6 shows data on bee populations from the USDA. However, it is not of bee populations, but of bee colony deaths, where a decline in values is actually good news. It would be nice to fit a line to these data to find out the actual rate of decline.

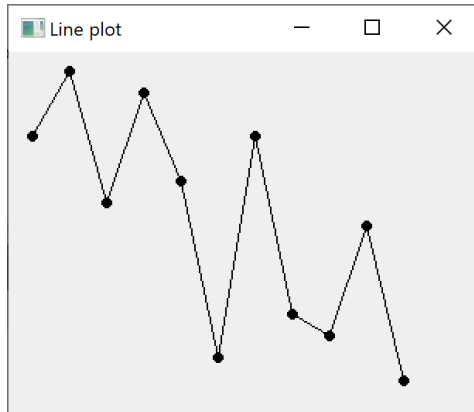


Figure 15-6 - Data showing declining bee colony deaths over time.

To fit a set of x-y data points to a curve, you specify a **colvec** for x and another for the y data, and the order of the polynomial to fit the data. Polyfit returns the coefficients for a n-order polynomial in the form:

$$y = p_0x^n + p_1x^{n-1} + p_2x^{n-2} + \dots + p_{n-1} + p_n$$

For a straight line, which has the form

$$y = mx + b$$

Or in this case,

$$y = p_0x^1 + P_1$$

This is thus a first order polynomial and the function call looks like this:

```
vec p = polyfit(ary, ary, 1);
```

The vector **p** then contains two elements equivalent to the slope and intercept, or *m* and *b*. For the data on bee populations, you then calculate *y* at x_0 and at x_{n-1} . The fitted line looks like this:

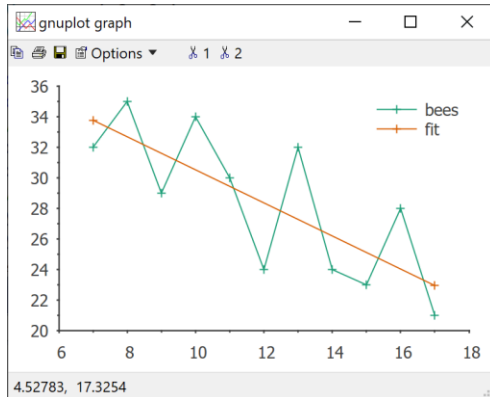


Figure 15-7 - Fitted plot of bee colony deaths, 2007-2017

This final plot was drawn with Sciplot, and we'll explain it in the following chapter.

Installing and running Armadillo programs

You can download the Armadillo tar.xz file from its sourceforge website¹ and unzip it in the usual way. For all platforms, consult the README.MD file in the armadillo root directory. For Windows, this is most of the job, but you must create the armadillo library by changing to the outer directory which might be something like `c:\armadillo-11.2.4`, and typing

```
Cmake .
```

This invokes the Cmake build package to create the library files. Be sure that you type the dot separated from the Cmake command by one space.

Running the example program

If you are using Visual Studio on Windows, you will find under the examples folder the files

```
Example1.cpp
example1_win64.sln
and
```

example1_wind64.vcxproj

If you open the .sln solution file with Visual Studio, and run the example, it will run a whole list of simple armadillo commands in a console window. You can add or modify this example code to see what some of the other functions and classes do.

Running new armadillo programs

In most cases, you can create a windows project, or any other platform, by setting your compiler to find the armadillo include files at `c:\armadillo 11.x.y\include` and the library files at `c:\armadillo\examples\lib_win64`.

There are a few cases where the linker will fail, and **polyfit** is among them. For such cases, create a new, empty project directory wherever Visual Studio is keeping projects. This is usually

`C:\Users\your-name\source\repos`

Copy the above three files into that new directory, open it using Visual Studio, and edit the `example1.cpp` to create the program you want to run. Rename it to whatever you want to call it and run it.

Example programs on GitHub

1. `ArmSimp.cpp` – simple examples used in this chapter
2. `Armamatrix.cpp` – Matrix transpose display
3. `SineFFT` – creates decaying sine wave and performs FFT, displays both
4. `Fitbees` – plots bee colony failure data by year and fits straight line to it.

References

1. <https://arma.sourceforge.net/>
2. <https://arma.sourceforge.net/docs.html>

3. Conrad Sanderson and Ryan Curtin.
Armadillo: a template-based C++ library for linear algebra.
Journal of Open Source Software, Vol. 1, pp. 26, 2016.
4. Conrad Sanderson and Ryan Curtin.
A User-Friendly Hybrid Sparse Matrix Class in C++.
Lecture Notes in Computer Science (LNCS), Vol. 10931,
pp. 422-430, 2018.
5. Morten Hjorth-Jensen, Data analysis and Machine Learning Lectures: Linear Algebra, Handling of Arrays.
<https://mhjensen.github.io/MachineLearningMSU-FRIB2020/doc/pub/Linalg/html/Linalg.html>
6. Jon Entine, Beepocalypse Myth Handbook, *Genetic Literacy Project*, <http://bit.ly/3WwmUiN>

16. Plotting in C++

In Python, you can use the Matplotlib library to plot data in a variety of ways. In Linux and the MacOS, there is an interface to Matplotlib that requires that you have Python installed as well, but there is no such version for Windows. So, in this chapter, we'll take a look at three methods of plotting data that work very well.

You can create plots of your data in C++ in any number of ways. For example, you could plot that honey bee data yourself using the wxWidgets library. The simplest way is just to create a Frame and draw in it. The only tricky part of drawing in wxWidgets is that you have to create a Paint event handler that redraws the screen whenever necessary. So, part of your constructor method is connecting to the paint event handler, which is called whenever the program needs to redraw the screen:

```
//frame constructor
LinePlot::LinePlot(const wxString& title)
    : wxFrame(NULL, wxID_ANY, title, wxDefaultPosition) {
    SetBackgroundColour(0xefefef);
    SetSize(WIDTH, HEIGHT);          //set window size

    //paint event is called to redraw window
    this->Connect(wxEVT_PAINT,
                wxPaintEventHandler(LinePlot::OnPaint));
    this->Center();
}
```

The paint event handler, then becomes the place where you do all your drawing.

Doing all this plotting yourself is kind of a fussy job: you have to create an array of points and find the x and y max and min to calculate the scale factors to convert the data points to pixel positions yourself. We also allow a 5% border all around, so there is an x-edge and y-edge value to compute as well.

We start by creating a vector of **wxPoint** objects, each of which has a public **x** and **y** method to get the actual data value:

```
vector <wxPoint*> LinePlot::createData() {
    vector <wxPoint*> points;
    // create vector of x,y point coordinates
    points.push_back(new wxPoint(7, 32));
    points.push_back(new wxPoint(8, 35));
    points.push_back(new wxPoint(9, 29));
    :
    :
    findBounds(points);    //calculates the scale
    return points;
}
```

We then have routines **xscale** and **yscale** that use those scaling factors to convert the data points to pixels:

```
//convert data point to x pixel position
int LinePlot::calcx(double xval) {
    int newx = (xval - xmin) * xscale + xedge;
    return newx;
}
```

With this in mind, our Paint event handler is shown below. You do your drawing using a **wxPaintDC** *device context*. This code illustrates the one for plotting on a screen. It is possible to use the same code with other device contexts to create hard copies or Postscript files.

```
//all the work happens in the Paint event
void LinePlot::OnPaint(wxPaintEvent& event) {
    wxPaintDC dc(this);
    wxSize sz = this->GetSize();
    width = sz.GetWidth();
    height = sz.GetHeight();
    bheight = height - 25;    //height of title bar

    dc.SetBrush(*wxBLACK_BRUSH);    //fill color
    //dc.SetPen(wxPen(wxColor(0,0,255), 1)); //blue color
    vector <wxPoint*> pts = createData();

    // draw lines between points and add marker circles
    //get first point outside loop
```

```

wxPoint* p = pts[0];
int x1 = calcx(p->x);    //fetch and scale first point
int y1 = calcy(p->y);
dc.DrawCircle(x1, y1, 3); //draw the first marker

for (int i = 1; i < pts.size(); i++) {
    wxPoint* p1 = pts[i]; //fetch and scale point
    int x2 = calcx(p1->x);
    int y2 = calcy(p1->y);
    dc.DrawCircle(x2, y2, 3);    //Draw the marker
    dc.DrawLine(x1, y1, x2, y2); //and the line
    x1 = x2;    //copy current point to starting
    y1 = y2;
}
}

```

The resulting plot of this bee data at first looks like the data in Figure 16-1



Figure 16-1 - Upside down plot



Figure 16-2 - Right side up plot

However, we forgot that the screen coordinates have (0, 0) in the *top* left corner, while we need them in the lower left corner. We can easily correct this, in the **calcy** routine:

```

int LinePlot::calcy(double yval) {
    int newy = bheight * 0.8 - (yval - ymin) *
                yscale + yedge;
    //int newy = (yval-ymin) * yscale + yedge;
    //incorrect
    return newy;
}

```

Plotting using DrawLines

PlotIn the above example, we loop through the data, calculating the x and y scale and plot each point. The wxWidgets package

also allows you to plot the entire array in a single command using the **DrawLines** method:

```
dc.SetBrush(*wxBLACK_BRUSH);           //fill color
wxPoint* pts = createData1();

// draw lines between points and add marker circles
dc.DrawLines(DATASIZE, pts);

for (int i = 0; i < DATASIZE; i++) {
    //and 3 pixel circle at each point
    dc.DrawCircle(pts[i], 3);
}
```

This looks appealing, but it requires you to convert all the points to the pixel scale in advance and store them in a fixed size array. This is a bit clumsier than using a vector, so you may find it more trouble than it's worth.

So, as you can see, it is certainly possible to do quick and dirty plotting just using the wxWidgets GUI, but it would be better if we could use a plotting package where that work had already been done. Of course, Python has very good plotting library built into the language, called Matplotlib. It would be nice if this were accessible from C++, and there is such a library which calls the Python Matplotlib, called Matplotlibcpp. It is available for Linux and the Mac, but, unfortunately not for Windows. And requiring that Python be available on your system as well may be a drawback in production code.

There is also a project called Matplotlibplusplus created by Alan Defreitas, which you can download from GitHub, and which has a great deal of plotting functionality but a very steep learning curve, where you must learn enough about the CMake package to modify several Cmakelist.txt files that control which parts of the interrelated packages you need to compile. Documentation and support are quite limited.

SciPlot

One of the best plotting packages for you to use in C++ is SciPlot. SciPlot is really just a bunch of header files, because under the covers, SciPlot calls the public domain gnuplot package. Thus, you must install gnuplot as well as SciPlot. The online documentation provides a number of tutorials illustrating 2D and 3D plotting.

It was not at all difficult to adopt one of the tutorials to plot our fitted bee data as shown in Figure 16-3. Once you install the SciPlot libraries, you must reference the SciPlot include files in your program, but the coding is straight forward. Note that the last line of the program writes out a PDF file of the plotted data.

To use SciPlot, you need to understand their `Vec` variable type. It is actually just a short alias for the `std::valarray` class. This class behaves much as a vector of doubles does, but `valarray` supports element-wise mathematical operations and convenient forms of indexing and slicing. We don't need any of those in this simple example, where it works like an array.

```
#include <sciplot/sciplot.hpp>
using namespace sciplot;

int main(int argc, char** argv) {
    // the bee plotting data
    Vec x = { 7,8,9,10,11,12,13,14,15,16,17 };
    Vec y = { 32,35,29,34,30,24,32,24,23,28,21 };

    // xy pairs describing the fitted line
    Vec xa = { 7,17 };
    Vec ya = { 33, 22.94 };

    // Create a Plot object
    Plot2D plot;

    // This disables the deletion of
    //the created gnuplot script and data file.
    plot.autoclean(false);

    // Plot the data and the fitted line
    plot.drawCurveWithPoints(x, y).label("bees");
}
```

```

plot.drawCurveWithPoints(xa, ya).label("fit");

// Create figure to hold plot
Figure fig = { {plot} };

// Create canvas to hold figure
Canvas canvas = { {fig} };
canvas.size(800, 500);

// Show the plot in a pop-up window
canvas.show();

// Save the plot to a PDF file
canvas.save("beedata.pdf");
}

```

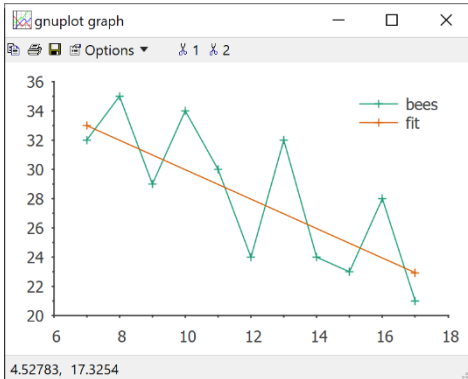


Figure 16-3- Bee data plotted in SciPlot

You can, of course, write a program that uses `armadillo` for the curve fitting we discussed in the previous chapter and plot the result as well. The only tricky part is that `armadillo` uses arrays called `vec` which are totally different from SciPlot's `Vec` arrays, and there is no simple conversion between them. Instead, you have to start with the data in `vec` arrays and after doing the fitting, make `Vec` arrays for SciPlot to use.

That is actually quite simple as we see here:

```

//compute the two y values
//for the first and last x points
Vec xa = { 0, 0 };
Vec ya = { 0, 0 };

//calculate two points on the fitted line
double y0 = m * x[0] + b;
double ylast = m * x[size(x) - 1] + b;
//the two x points
xa[0] = x[0];
xa[1] = x[size(x) - 1];
//the two y points
ya[0] = y0;
ya[1] = ylast;

//draw fitted straight line on graph
plot.drawCurveWithPoints(xa, ya).label("fit");

```

ROOT

ROOT is a very sophisticated plotting package developed at CERN for plotting physics data. ROOT can do standard xy plots as well as some extremely powerful surface plots.

You can download precompiled versions of ROOT for nearly all platforms, although the Windows version is nominally still in beta. We have been using it without difficulties, however. Once downloaded, it is a good idea to make ROOT part of your PATH, so you can access it from anywhere on your computer. It is also helpful to create an environment variable ROOTSYS the points to the base directory of the package. You can use this variable to define where the include and library files are located when you create Visual Studio projects. You can then add `$(ROOTSYS)include` to your include directories and `$(ROOTSYS)lib` to your library directories of your project.

You can use ROOT in an interactive mode to plot data, or you can write simple C++ programs that link to ROOT as well. In the interactive mode, you start the ROOT command and tell it to run little plotting programs. ROOT contains a C++-language

interpreter for this step, so you don't need to compile and link every time you make a change.

The ROOT interpreter

The easiest way to familiarize yourself with ROOT, is by using the interpreter. Then building the C++ code for the final product gets much easier.

Start by selecting a working directory for your trials anywhere on your computer, and use the `plotdata1.txt` file we provide, or make your own. You just need a list of pairs of `x` and `y` values on separate lines, like this:

```
# trial data plotdata1
#x    y
1.0   1.0
3.0   2.0
3.0   3.5
5.0   3.0
5.0   5.0
7.0   5.0
7.0   7.0
8.0   6.0
```

Then, start the ROOT interpreter, by just typing

```
Root
```

in a CMD window. This will give you the prompt

```
root[0]
```

There are a number commands you can give the ROOT interpreter. Each of them begins with a dot: the three most important ones are:

- `.q` - exit from the interpreter
- `.L` - load a C code file
- `.x` - execute a C code file

First, we are going to create a TGraph object containing the above data by typing:

```
TGraph gr("plotdata1.txt")
```

This creates a TGraph object, which holds one set of plotting data. You can then tell it to plot the data by typing:

```
gr.Draw("LA")
```

This says to draw a line and the axes around it as shown in Figure 16-4

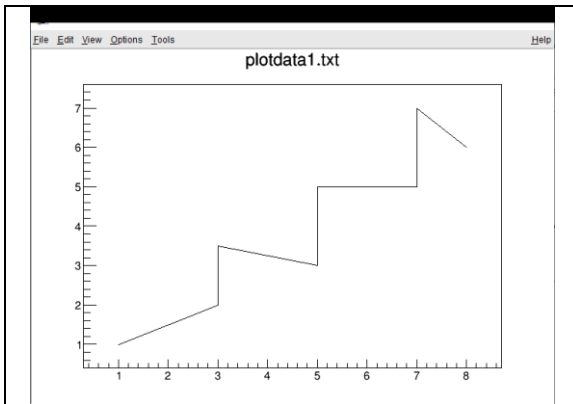


Figure 16-4 -- ROOT line plot

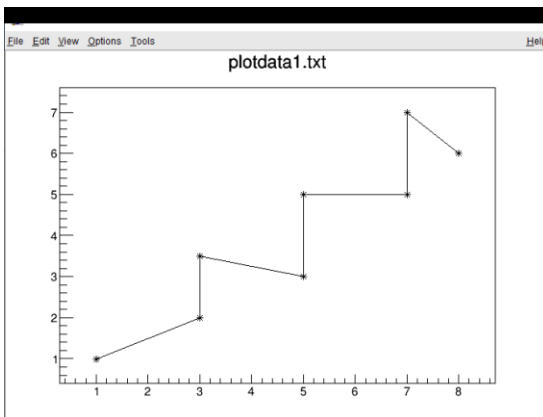


Figure 16-5 -- ROOT line plot with * markers

If you add an asterisk to the command,

```
gr.Draw("LA*")
```

it draws the line with asterisks for each data point as shown in Figure 1-5. You have a choice of a wide variety of markers that you can define by number. For example, if you enter

```
gr.SetMarkerStyle(20)
gr.Draw("PLA")
```

you will get a plot with solid circles, (Figure 16-6). The command letter “P” means plot using the current marker style.

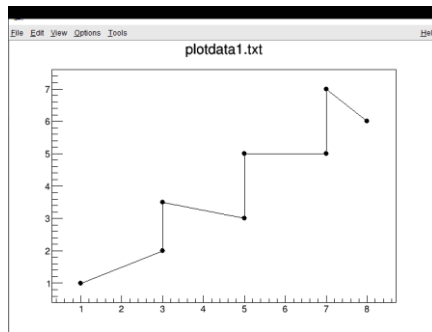


Figure 16-6 -- Marker style 20, solid circle

There are dozens of marker styles, a few of which we show in Table 16-1.

1	Dot
2	Plus
4	Open circle
9-19	Large dot of various sizes
20	Full circle
21	Square
22	Triangle up
23	Triangle down

Table 16-1- Marker style values in ROOT

The complete table is shown in Reference 9.

Likewise, there are a large number of character modifiers to the Draw method [217], some of which are shown in Table 16-2.

A	Axis
L	A simple polyline
*	A star is drawn at each point
P	The current marker is drawn at each point
B	Draw a bar chart
I	With B, makes bar start at bottom instead of 0

Table 16-2 - Draw method modifiers.

Writing C++ code for ROOT

You can, of course write complete C++ programs to generate your plots, and save them as files. If you write code to run in the ROOT interpreter, your C++ code must start with the needed include files for ROOT, and contain a single function that has the same name as the file itself. So the file `plotlines.C` contains a single function named `plotlines()` as shown below:

```
#include "TF1.h"
#include "TApplication.h"
#include "TCanvas.h"
#include "TRootCanvas.h"
#include "TGraph.h"
#include "TLegend.h"
#include "TFont.h"
#include "TPad.h"

// Draw a simple x-y plot
void plotlines() {
    auto c = new TCanvas("c1", "Demo example");
    auto gr = new TGraph("plotdata1.txt"); //read in file
    gr->SetTitle("Demo data"); //title on the plot
    gr->SetMarkerStyle(20); //filled circle marker
    gr->Draw("APL"); //plot axis and lines
    return 0;
}
```

As you can see, the function **plotlines** creates a canvas and puts the text in title bar. It creates the TGraph object, which reads in the file `plotdata1.txt`, creates a title, sets a marker type and draws the plot.

To use this code you can simply load the program using the “.L” command and run the function:

```
.L plotlines.C
plotlines()
```

Or, you can execute the file in a single step by using the “.x” command:

```
.x plotlines.C
```

You can also run this function from the CMD command line without starting the ROOT interpreter directly by typing

```
root -x plotlines.C
```

Writing ROOT code for a C compiler

If you want to actually compile and run this program as a stand-alone executable, you simply change the name of the function to **main()**. Then, the code looks like this:

```
#include "TF1.h"
#include "TApplication.h"
#include "TCanvas.h"
#include "TRootCanvas.h"
#include "TGraph.h"
#include "TLegend.h"
#include "TGFont.h"
#include "TPad.h"

// Draw a simple x-y plot
int main(int argc, char** argv) {
    TApplication app("app", &argc, argv);
    auto c = new TCanvas("c1", "Demo example");
    auto gr = new TGraph("plotdata1.txt"); //read in file
    gr->SetTitle("Demo data");           //title on the plot

    gr->SetMarkerStyle(20); //filled circle marker
    gr->Draw("APL");        //plot axis and lines
    c->Print("demo.pdf");   //save PDF of plot
    app.Run();              //run the plotting window
    return 0;
}
```

Note that this code is a console program, not a Windows program, and is therefore much easier to write. ROOT creates windows for the resulting plots from your console application. Note that you can save a PDF of the plot with a single statement. There are about a dozen or so possible output file formats that you can choose (such as ps, png, and jpeg) simply by varying the file extension [10]. And finally, note that you have to create a **TApplication** object, and run it when all the graphics have been created.

In Visual Studio, you need to make sure that the include and library paths are set as we described above, and that the conformance mode is set to “no /permissive.”

Error bars

You can easily create x-y plots that include error bars in both the x and y directions. TO do this, you expand the data file to include columns for the x-error and y-error of each data point. In this example we made the x-errors zero so that the y errors stand out. We also reduce the point marker to a single small dot to reduce visual confusion. The error data file looks like this:

```
# trial data plotdataerr1
#x          xerr yerr
1.0        1.0  0     .2
3.0        2.0  0     .1
3.5        3.5  0     .3
5.0        3.0  0     .05
5.5        5.0  0     .1
7.0        5.0  0     .14
7.5        7.0  0     .2
8.0        6.0  0     .3
```

In the code, we simple change the graphic object to TGraphErrors instead of TGraph:

```
auto gr = new TGraphErrors("plotdataerr1.txt");
```

The program is otherwise unchanged. The resulting error plot is shown in Figure 16-7.

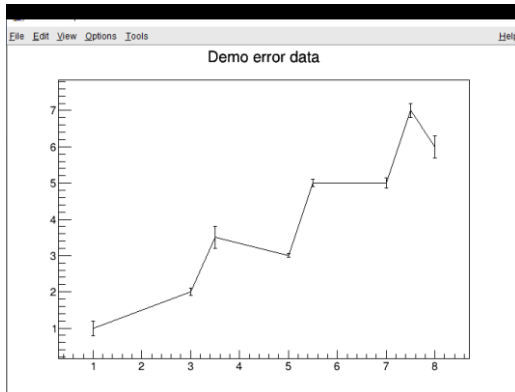


Figure 16-7 -Error bars in y direction

Plotting multiple lines in ROOT

To plot the bee die-off data in ROOT, we create three objects: a TGraph for the bee data, a TGraph for the fitted line, and a TLegend box for the line color and marker legend. Note that we plot the Bee data including the axes (AL*), but we plot the fitted line without redrawing the axes (LP). The colors are set using the constant names shown in the TColor description [11].

You create a legend box using a size that is the fraction of the size of the window, here 0.2 by 0.3. You then add the two entries using the relevant TGraph variable and the text describing it. The legend box location is computed when the data are drawn. However, you can drag the legend box to a new position using your mouse before saving a hard copy.

The complete code for plotting the bee data is:

```
#include "TF1.h"
#include "TApplication.h"
#include "TCanvas.h"
#include "TRootCanvas.h"
#include "TGraph.h"
#include "TLegend.h"
#include "TGFont.h"
```

```

#include "TPad.h"

int main(int argc, char** argv) {
    TApplication app("app", &argc, argv);

    //the bee data
    double x[11] = { 2007,2008,2009,2010,2011,
                    2012,2013,2014,2015,2016,2017 };
    double y[11] = { 32, 35, 29, 34, 30,
                    24, 32, 24, 23, 28, 21 };

    //create bee population die-off graph
    TGraph* gr1 = new TGraph(11, x, y);

    gr1->SetLineColor(kBlue + 1);
    gr1->SetTitle("Bee population die-off");
    gr1->Draw("AL*");

    // create the line that fits the data
    double xfit[2] = { 2007,2017 };
    double yfit[2] = { 33, 22.9 };
    TGraph* gr2 = new TGraph(2, xfit, yfit);
    gr2->SetLineColor(kOrange );
    gr2->SetMarkerStyle(21);
    gr2->SetMarkerColor(kOrange );
    gr2->Draw("LP");

    // create the legend
    TLegend* legend =
        new TLegend(0.2, 0.1, "", "brNDC");
    legend->AddEntry(gr1, "bee data");
    legend->AddEntry(gr2, "bee fit");
    legend->Draw();

    app.Run();    //run the plotting window
    return 0;
}

```

And the actual plotted data is shown in Figure 16-8.

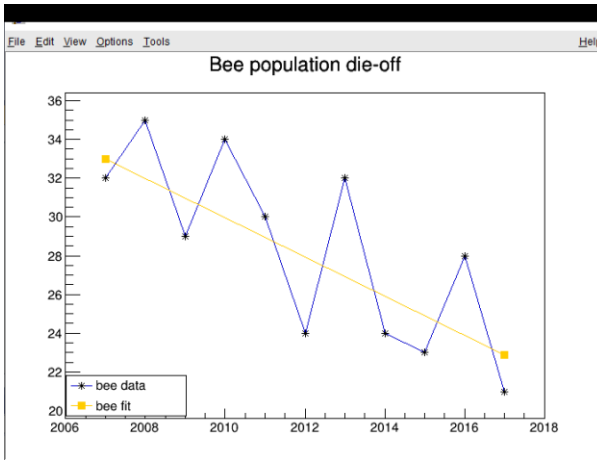


Figure 16-8 - ROOT plot of bee die-off data

Example programs on GitHub

1. wxBeePlotVector – plotting bee data from a vector
2. wxBeePlotArray – plotting using an array and DrawLines
3. splottest – Plotting using SciPlot
4. fitbees – curve fitting of bee data combined with SciPlot plotting
5. plotlines.C – Plotting simple x-y data using ROOT
6. plotlineserr.C – Plot lines with error bars
7. RootTestPlot – Plotting same fitted bee data using ROOT

References

1. wxWidgets drawing -- https://docs.wxwidgets.org/3.1.4/classwx_d_c.html
2. Matplotlib for C++ -- <https://matplotlib-cpp.readthedocs.io/en/latest/>

3. Matplotlib --
<https://alandefreitas.github.io/matplotlibplus/>
4. SciPlot -- <https://sciplot.github.io/>
5. Installing SciPlot-- <https://sciplot.github.io/installation/>
6. GnuPlot -- <http://www.gnuplot.info/>
7. Root reference -- <https://root.cern/doc/v626/index.html>
8. Root plotting options --
<https://root.cern.ch/doc/master/classTGraphPainter.html>
9. Root marker table --
<https://root.cern.ch/doc/master/classTAttMarker.html>
10. Root print file types --
<https://root.cern.ch/doc/master/classTPad.html>
11. Root colors -
<https://root.cern.ch/doc/master/classTColor.html>

17. Databases in C++

Relational databases are an important part of the way we store and connect data in both business and science. A relational database consists of a series of tables that are connected to each other using integer keys. To see how this works, let's look at the three tables in our groceries database. Here, the displays are from SQLite Studio.

	foodkey	foodname		storekey	storename
1	1	Apples		1	Stop and Shop
2	2	Oranges		2	Village Market
3	3	Hamburger		3	Shoprite
4	4	Butter			
5	5	Milk			
6	6	Cola			
7	7	Green beans			

Figure 17-1 -Food table

Figure 17-2 -Stores table

The tables in Figure 17-1 and Figure 17-2 show a table of foods and a table of stores, each with a number key column on their left.

Now, if we want to create a table of prices in each of the three stores, we wouldn't repeat the food names and store names. Instead, we'd simply refer to them by their key number. So line 1 of our price table shown in Figure 17-3 indicates the Apples (1) at Stop and Shop(1) cost \$0.27 each.

	pricekey	foodkey	storekey	price
1	1	1	1	0.27
2	2	2	1	0.36
3	3	3	1	1.98
4	4	4	1	2.39
5	5	5	1	1.98
6	6	6	1	2.65
7	7	7	1	2.29

Figure 17-3 - Price table

While these are the names of real stores, the actual prices are entirely fictional.

If you wanted to generate a report of the prices by food or store, you use the SQL (Structured Query Language) of the database to create the report. For example,

```
select foodname, storename, price from
foods, stores, prices
where foods.foodkey= prices.foodkey and
stores.storekey = prices.storekey
```

will give you a list of the foods and prices at each store: some of them being:

```
Apples      Stop and Shop    0.27
Oranges     Stop and Shop    0.36
Hamburger   Stop and Shop    1.98
Butter      Stop and Shop    2.39
Milk        Stop and Shop    1.98
Cola        Stop and Shop    2.65
Green beans Stop and Shop    2.29
```

Although the complete result has 21 items in it: seven for each of three stores. You can also sort these differently by adding

```
order by foodname, price
```

to the end of the query, giving you in part

```
Apples Stop and Shop 0.27
Apples Village Market 0.29
Apples Shoprite 0.33
Butter Stop and Shop 2.39
Butter Village Market 2.99
Butter Shoprite 3.29
Cola Stop and Shop 2.65
```

These are, of course, relatively simple examples of using a database.

SQLite

The SQLite database is an easy program to start with. It runs on any platform and will probably run on your laptop or desktop without much fuss. Each database SQLite uses is a single file you tell it to open. It doesn't have the overhead of some complex client server process. Our little groceries database is called **groceries.db** and is only 28 kb in size.

According to the SQLite web pages, SQLite is used on most mobile phones, and can be used in games, televisions, cameras and watches. It is also suitable for use on medium traffic web sites. It is designed to be a zero maintenance database. On the other hand, for large multiuser databases, you probably should use MySQL or one of the major commercial database products.

Downloading SQLite

You can download SQLite from their website: versions are available for just about all computers and operating systems. In the case of Windows, choose the 64-bit version, and unzip the file into a convenient location such as C:\sqlite64 or under Program Files\sqlite64.

Once you have installed SQLite, you should set your PATH to include its root directory. Then you can type the command "sqlite3" and play with the command line interpreter. Most of its commands start with a dot. So you could go to a directory where

you have a copy of our groceries.db database (downloadable from our GitHub site).

Then you could type

```
Sqlite3
.open groceries.db
.tables
```

And get a list of the tables in this database.

```
foods  prices  stores
```

But these line commands can quickly be tedious. You will do a lot better downloading SQLiteStudio and running it.

SQLiteStudio

SQLiteStudio is a convenient cross-platform windowing interface for viewing and changing the contents of an SQLite database.

Once you start it, you can open any existing database or create a new one.

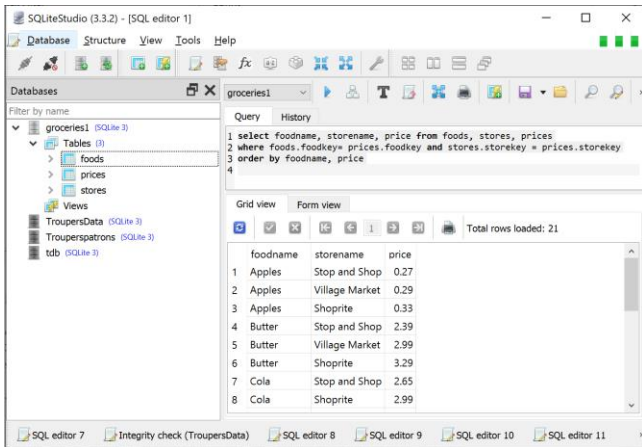


Figure 17-4 SQLiteStudio window showing groceries database

In Figure 17-4, you see the window, showing the tables and showing the results of an SQL query. This is a good place to practice writing SQL statements before inserting them into any program you may develop.

Programming SQLite in C++

To program in C++ to the SQLite interface,

1. you need to download the source code for SQLite and snag a copy of the header file **sqlite3.h**. Put this in the root directory of your sqlite3 program, or as we did, in a subsidiary **\include** directory under C:\sqlite4.
2. You also need to create an **sqlite3.lib** file. In that same root directory, you will find a file called **sqlite3.def**. You can use it to create that lib file by typing in a command window:

```
LIB /DEF:sqlite3.def /MACHINE:X64
```

Compiling using Visual Studio

Download the first example program **sqlitest.cpp** and create a console project with that same name copy or paste this .cpp file into your project.

1. You must add c:\sqlite64 to your project VC++ include path and to the library directories as shown in Figure 17-5.

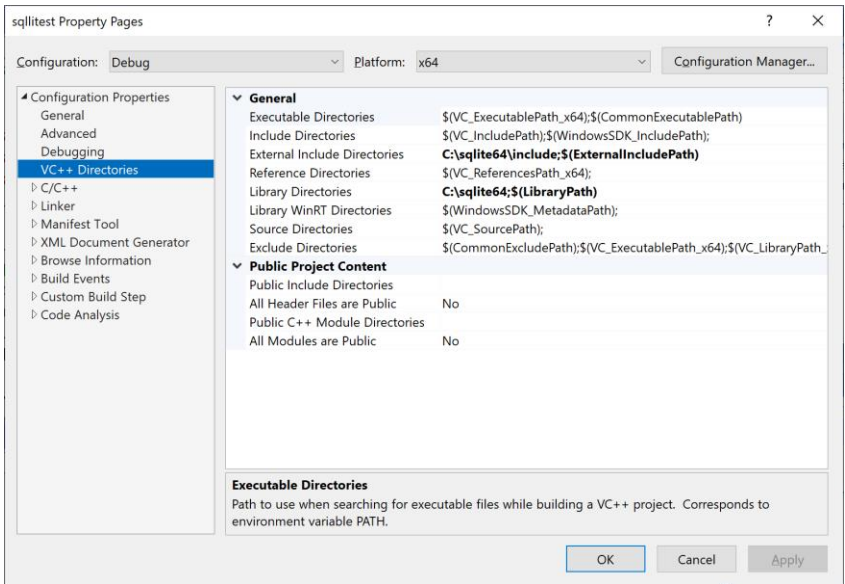


Figure 17-5 – Adding *sqlite64* directory to include and library paths

2. Add `c:\sqlite64` to the linker “Additional Library Directories” as shown in Figure 17-6.

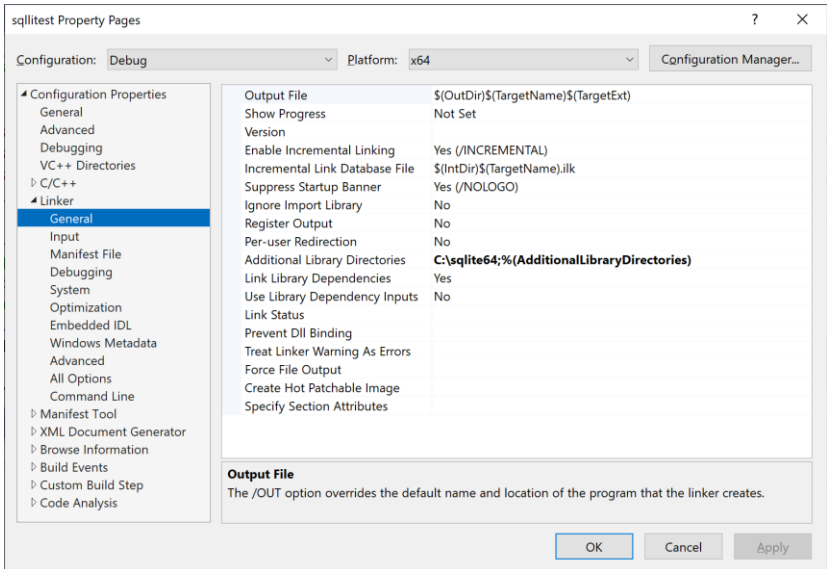


Figure 17-6 - Add *sqlite3* directory to the Linker's Additional Library Directories

3. You also need to add a reference to the LIB file we created earlier. This goes under Linker Input/Additional dependencies. See Figure 17-7.

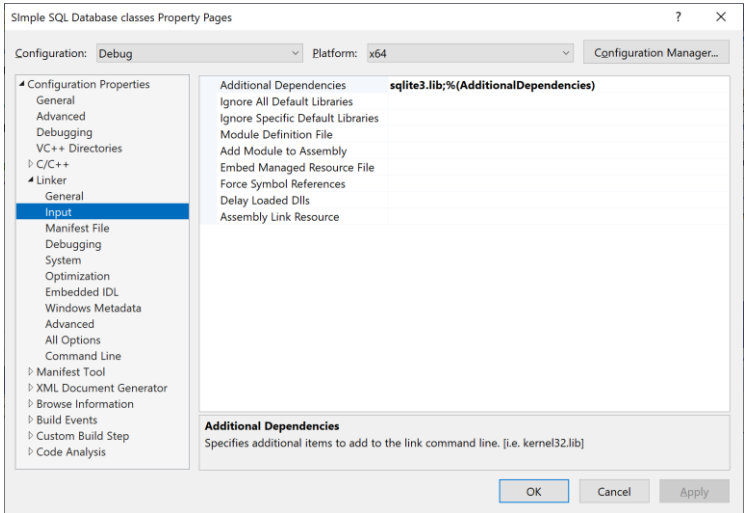


Figure 17-7 - Adding the sqlite3.lib file

4. Your Visual Studio Project folder will now look like this:

sqlitest			
	.vs		
	sqlitest		
		x64	
		groceries.db	
	X64		
		Debug	
			sqlite3.dll

Copy the groceries.db file into sqlitest\sqlitest and copy the sqlite3.dll files into sqlitest\x64\Debug. The dll file must be in the same directory where the compiled sqlitest.exe is put. This way the runtime system will find that dll.

If you are uncomfortable with copying that DLL all over the place while you writing programs for SQLite, you can instead go Properties/Configuration/Debugging/Environment and add the statement

`PATH=c:\sqlite64;%PATH%`

This is shown in Figure 17-8.

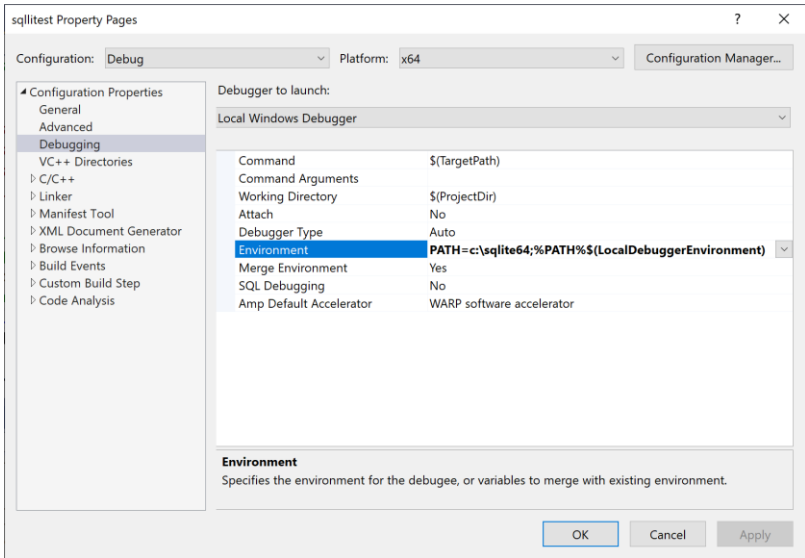


Figure 17-8 --Inserting `PATH` for runtime to find the `SQLite3.dll`

Example C++ code to connect to SQLite

The basic code to connect to an SQLite database starts with opening a connection to a file:

```
sqlite3* db;
int rc;

/* Open database */
rc = sqlite3_open("groceries.db", &db);
```

Then, you enter a query by calling the `sqlite3_exec` function:

```
char* zErrMsg = 0;
const char* sql;
const char* data = "Callback function called";
```

```

/* Create SQL statement */
sql = "Select foodname, price, storename \
      from foods, prices, stores \
      where foods.foodkey = prices.foodkey \
      and stores.storekey = prices.storekey \
      order by foodname, price";

/* Execute SQL statement */
char* zErrMsg = 0;
const char* sql;
const char* data = "Callback function called";
rc = sqlite3_exec(db, sql, callback,
                 (void*)data, &zErrMsg);

```

Then, what happens is that the SQLite code calls the **callback** function, which is where the results come back. It is called for each row the query returns. In this example, the callback prints out each row. `argc` indicates the number of elements in the row, which should match the number in the SELECT statement.

```

// the callback routine that SQLite calls for each row
int callback(void* data, int argc, char** colVal, char**
colName) {
    // each callback returns one row
    for ( int i = 0; i < argc; i++) {
        //print each value in row
        std::cout << " " << colVal[i] << " ";
    }
    std::cout << std::endl; //end line for each row
    return 0;
}

```

The result printed out looks, in part like this:

```

Opened database successfully
Apples 0.27 Stop and Shop
Apples 0.29 Village Market
Apples 0.33 Shoprite
Butter 2.39 Stop and Shop
Butter 2.99 Village Market
Butter 3.29 Shoprite
Cola 2.65 Stop and Shop
Cola 2.99 Shoprite
:
Operation done successfully

```

Building a database class structure

If you think about what classes you might need to use to connect with a database, you probably would come up with

- Database
- Query
- Results

at the very least. For SQLite, our database class is pretty simple: it does little more than wrap the `sqldb` object:

```
//The basic database object
//contains a pointer to the sqlite database
class sqltDatabase {
private:
    sqlite3* sqldb{ NULL };
    string dbname;

    bool exists(const std::string& name) {
        ifstream f(name.c_str());
        return f.good();
    }
public:
    int sqltDatabase::open(std::string fname) {
        /* Open database -- error if it doesn't exist */
        dbname = fname;
        rc = sqlite3_open_v2(dbname.c_str(), &sqldb,
            SQLITE_OPEN_READWRITE , NULL);
        return rc;
    }
    int sqltDatabase::create(std::string fname) {
        /* Open database or create it */
        dbname = fname;
        rc = sqlite3_open_v2(dbname.c_str(), &sqldb,
            SQLITE_OPEN_CREATE |
            SQLITE_OPEN_READWRITE, NULL);
        return rc;
    }
    //return the sqlite pointer
    sqlite3* getDb() {
        return sqldb;
    }
}
```

```
//close the database connection
    void close() {
        sqlite3_close(sqldb);
    }
};
```

If you use the original `sqlite3_open` function to open a database file that doesn't exist, it creates an empty file in that directory having that name. This is while we use the `sqlite3_open_v2` function instead.

The Query object

Most of the work in getting data from our database takes place in executing the query. In SQLite, we additionally have to deal with that callback structure we illustrated in the `sqlitest` example above. Here, we wrap all of that in the `query`.

In addition, we have to decide what form the query results should take. In this implementation, we decided to use a map for each value returned. That way you can get the results out in any order, regardless of the order you chose in the SQL itself.

So for thee query

```
Select foodname, price, storename.. .
```

We return each new row as a map. For example, this is the map returned for the first result of that query:

```
{
    {"foodname", "Apple"},
    {"price", "0.27"},
    {"storename", "Stop and Shop"}
}
```

To simplify use of these maps, we create an abbreviation where we name that map as a **dbMap**.

```
//convenient abbreviation for the map
typedef map<string, string> dbMap;
```

The critical part of the Query class is the callback method, which must be **static** for the callback from SQLite to work. It keeps two static variables, one where each row's map is created, and one which is a vector of those successive maps.

```
class Query {
public:
    static dbMap stRow; //the map accrues here
    static vector <dbMap> stRows; //vector of maps
```

The callback function is called by SQLite for each row of the query result. Each call contains arrays of names and values which we put into a map entry. When all of the values in that row have been added to the map, that map is added to a vector of rows:

```
static int callback(void* data, int count,
                   char** colVal, char** colName){
    Query::stRow.clear(); //clear the map variable
    for (int i = 0; i < count; i++) {
        string s1 = string(colName[i]); //name
        string s2 = string(colVal[i]); //value
        //insert into map
        Query::stRow.insert({ s1, s2 });
    }
    //add map to vector
    Query::stRows.push_back(Query::stRow);
    return 0;
}
```

The actual Query **execute** method makes the call to SQLite and passes it the pointer to the static callback method.

```
// execute the query:
// the results are assembled
// in the static callback function
// as a vector of maps, one for each line of the results
// a pointer to a Results object is returned,
// containing that vector
```

```
Results* execute() {
    int rc = sqlite3_exec(sqdb, qstring.c_str(),
        Query::callback,
```

```

        nullptr, &errMsg);
    return new Results(stRows);
}

```

When SQLite finishes with the callbacks, control is passed to the return statement, which returns a Results object containing that vector.

The Results class

The Results class copies that static vector from the Query class into an instance variable and then provides a number of ways for you to access that data. The most common ones are to get the whole map row, or to get one column by value. Results also increments the cursor each time, so you always can get the next row.

```

//copy from the Query static vector
Results::Results(vector<dbMap> crows) {
    for (int i = 0; i < crows.size(); i++) {
        rows.push_back(crows[i]);
    }
}
//get one row as a map
dbMap Results::getRow() {
    return rows[cursor++];
}
//get the value of the named column in the current row
string Results::getVal(string name) {
    return rows[cursor++][name];
}
//get the size of the result
size_t Results::getSize() {
    return rows.size();
}
}

```

Using the SQLite classes

You can see now, that calling these classes is much easier than using the original example code. Here is all it takes:


```

//program starts here
int main(int argc, char* argv[]) {
    //open the database
    SQLiteDatabase db("groceries.db");

    // create the query
    //note the continuation characters within a string
    Query qry(db,
        "Select foodname, price, storename from prices \
        join foods on (foods.foodkey = prices.foodkey) \
        join stores on (stores.storekey = prices.storekey) \
        order by foodname, price");

    //execute the query
    Results* res =qry.execute();

    //print out results of query
    for (int i = 0; i < res->getSize(); i++) {
        dbMap r = res->getRow();
        cout << r["foodname"] << ": " << r["price"]
            << ": " << r["storename"] << endl;
    }
    return 0;
}

```

Database Tables

We have shown you examples of database tables already in our little grocery example. But so far, we haven't shown how you create a table. You can create a table in a single SQL command if you want. For example, to create the Foods table, you create a table with a key and a storename like this:

```

CREATE TABLE STORES
    (storekey INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    storename VARCHAR(45) NOT NULL )

```

But, as you can see, this is a little fussy to get right. Let's look into building a Table class that will make tables for us. The above example shows an integer key column and a variable length character column with a maximum length of 45. The only

other common column would be the float column (double is not available in most databases).

So before we consider making our table class, we might consider making a basic Column class.

```
class Column {
protected:
    string colname;           //the name
    bool primary{ false };   //and whether it's primary

public:
    Column(string name);
    bool isPrimary();
    string name();
    //derived classes fill this one in
    virtual string getSql()=0;
};
```

Note that this is an abstract class, since the `getSql` method is virtual, and will differ in each derived class.

A simple example for integer columns is

```
IntCol::IntCol(string name) :Column(name) {}
    string IntCol::getSql() {
        return colname + " INT NOT NULL ";
    }
```

Now, the primary key column is an integer as well, but you need to mark it PRIMARY and usually AUTOINCREMENT as well.

Here is that column's constructor and `getSql` method:

```
PrimaryCol::PrimaryCol(string name, bool auto_inc)
    :Column(name) {
    primary = true;
    autoinc = auto_inc;
}

string PrimaryCol::getSql() {
    string idname = colname +
        " INTEGER NOT NULL PRIMARY KEY ";
    if (autoinc) {
        idname += "AUTOINCREMENT ";
    }
```

```

    }
    return idname;
}

```

And, in the header file, we make sure that the `auto_inc` argument defaults to `true`.

```
PrimaryCol(string name, bool auto_inc = true);
```

The only slightly different column is the `CharCol` which creates a variable length text column, so you need to specify that size in the constructor:

```
CharCol::CharCol(string name, int cwidth) :Column(name) {
    width = cwidth; //character string width
}
string CharCol::getSql() {
    string idname = colname + " VARCHAR(" +
        to_string(width) + ") NOT NULL ";
    return idname;
}

```

Now, we can create the three tables quite easily. For example, here is the `Stores` table:

```
Table* stores = new Table("stores", db);
stores->addColumn(new PrimaryCol("storekey"));
stores->addColumn(new CharCol("storename", 45));
stores->create();

```

The `Foods` table is exactly the same.

The `Prices` table is a bit longer since it has 4 columns. For reasons we'll discuss in a minute, we actually create a derived `PriceTable` which creates the columns in its constructor:

```
addColumn(new PrimaryCol("pricekey"));
addColumn(new IntCol("foodkey"));
addColumn(new IntCol("storekey"));
addColumn(new FloatCol("price"));
create();

```

Adding rows to a Table

The SQL for inserting a row is pretty simple for small tables like the Stores:

```
insert into stores (storename) values ("Stop and Shop")
```

Note that **storename** is a column name and not a string, but “Stop and Shop” is a string. Again, this can get fussy, so for these two simple tables: Stores and Foods, we can create a simple `addRow` method: Note that we don’t have to specify the primary key column at all. It automatically increments with each additional row:

Our basic `addRow` method takes two strings, one with the column name and one a quoted string for the value:

```
stores->addRow("storename", "\"Stop and Shop\"");
stores->addRow("storename", "\"Village Market\"");
stores->addRow("storename", "\"Shoprite\"");
```

Note, however, that we have to use the `\` escape sequence to put a quote inside the quoted string. But for these simple tables, this isn’t too bad. For the Foods table there are seven such `addRow` calls for the seven foods.

But for the Prices table, it is a little more involved because you need to insert two integers and a float:

```
insert into prices (storekey, foodkey, price) values
(1,1,0.55).
```

These are two integer keys and a floating point price value. We need a way to represent a series of values which may be of several types: string, int and float. One obvious way is using a *tuple*. While you may be familiar with tuples where the values are all the same type, you can use the **make_tuple** function to create mixed tuples. For example

```
std::make_tuple(STOP, APPLES, 0.55)
```

returns a mixed tuple. But another way is to define a mixed tuple type:

```
typedef std::tuple<int, int, float> pricetuple;
```

We *could* create a special Table method for adding such rows:

```
price->addRow("storekey, foodkey, price",
pricetuple(1,1,0.55));
```

But it would be easier to derive a PriceTable class which contains this new version of the addRow method: The nice part is that when you call this method, you don't have to repeat the column names in each call. The beginning of this method creates the first part of the INSERT Sql.

```
int PriceTable::addRow( pricetuple ptuple) {
    string sql = "insert into " + tbName;
    sql += " (storekey, foodkey, price) ";
    sql += "values (" ;
```

Then in the next section we use C++'s **tie** function to unpack a tuple into 3 variables, and complete the SQL string:

```
int fkey; int skey; float price;
//unpack tuple into 3 variables
std::tie(fkey, skey, price) = ptuple;
sql += to_string(fkey) + ",";
sql += to_string(skey) + ",";
sql += to_string(price) ;
sql += ")";
```

While, we actually know what the key values will be for the stores and food table entries, we fetch them using a simple query that we wrote into the Table class:

```
const int STOP = stores->getKey("storekey",
    "storename", "Stop and Shop");
const int RITE = stores->getKey("storekey",
    "storename", "Shoprite");
const int VILLAGE = stores->getKey("storekey",
    "storename", "Village Market");
```

and similarly for the foods.

Then loading the price table amounts to this relatively simple code:

```
prices->addRow(std::make_tuple(STOP, APPLES, 0.55));
prices->addRow(pricetuple(STOP, BUTTER, 4.59));
prices->addRow(pricetuple(STOP, COLA, 8.95));
prices->addRow(pricetuple(STOP, BEANS, 3.49));
prices->addRow(pricetuple(STOP, BURGER, 6.95));
```

and so forth.

Prepared Queries

Sometimes we have to run almost the same query several times, where the only difference is one or two parameters in the WHERE clause. Taking that same grocery query, suppose just wanted a list of the prices of oranges in different stores. The query would be

```
select foodname, price, storename from prices \
    join foods on (foods.foodkey = prices.foodkey) \
    join stores on (stores.storekey = prices.storekey) \
    where foodname = "Oranges" order by price;
```

But later, we might decide to run a query on prices of butter, too. Clearly this is exactly the same query but with "Butter" replacing "Oranges" in the last line. It would be nice if we could create a generic query where you at the last minute decide which food to ask about.

A prepared query does just that. In that query, the last line becomes

```
where foodname = ? order by price
```

You create that query including one or more question marks replacing variable parameters and let the database compile it into internal byte codes. Then you simply tell the database what values to use for those parameters and run the query.

Code for carrying this out is very simple at the top level and looks much like previous code we've written.

```
db.open("mygroc.db");
string prepsql =
    "Select foodname, price, storename from prices \
    join foods on (foods.foodkey = prices.foodkey) \
    join stores on (stores.storekey = prices.storekey) \
    where foodname = ? order by price";

//create the prepared query
sqlite3_stmt* pq(db, prepsql);
pq.setVariable(1, "Oranges"); //set the variable 1
Results* res = pq.execute(); //run the query
```

You could, of course, have more than one variable. You refer to them by index, starting with one, in the order they are mentioned in the query. For example,

```
where (foodname = ?) or (foodname = ?) order by foodname,
price";
```

where you then set two variables as 1 and 2.

You print out the results in much the same way as before.

```
for (int i = 0; i < res->getSize(); i++) {
    dbMap dm = res->getRow();
    cout << dm["foodname"] << " " << dm["price"] << " "
        << dm["storename"] << endl;
}
pq.closeStatement(); //close and destroy the statement
```

The SQLite3 prepared statement interface operates in four steps: prepare the SQL statement (compile it), set variable values, step through the result rows, and finalize (delete) the statement. You also have the option to **reset** the prepared statement to the top of its compiled code, so you can run it again without recompilation.

Here is the `Sqlite3PrepQuery` constructor which carries out the preparation:

```

//create a prepared query-
//The "stmt" variable holds the pointer
//to the resulting prepared statement.
sqlitePrepQuery::sqlitePrepQuery(sqliteDatabase db,
    string query): Query(db, query) {
    int rc = sqlite3_prepare_v2(db.getDb(),
        query.c_str(),
        (int)query.size(), &stmt, NULL);
    checkerr(rc);
}

```

Then there are set variable methods for all the common types. The ones for int and double are quite simple:

```

//bind a double value to the query
void sqlitePrepQuery::setVariable(int index, double value) {
    int rc = sqlite3_bind_double(stmt, index, value);
    checkerr(rc);
}
//bind an int value to the query
void sqlitePrepQuery::setVariable(int index, int value) {
    int rc = sqlite3_bind_int(stmt, index, value);
    checkerr(rc);
}

```

The method for binding text variables is slightly more involved because the string variable that holds that string cannot be local to the method: it must be a class-level variable that has existence outside that method. This is because the string might be deleted before it is copied into SQLite. It is also important that you set the flag `SQLITE_TRANSIENT`, which tells the database that it should make a copy of that text before it gets deleted

```

//bind a text value to the query
void sqlitePrepQuery::setVariable(int index, string value) {

    //this MUST be a class level variable
    sval = value;
    const char* val1 = sval.c_str();
    size_t sz = strlen(val1);
    int rc = sqlite3_bind_text(stmt, index,
        val1, (int)sz, SQLITE_TRANSIENT);
    checkerr(rc);
}

```


You can then step through the rows, storing them in a vector of maps just as we did for the conventional queries:

```
//execute the query
// and get back the column names and values
Results* sqltPrepQuery::execute() {
    stpRows = new vector<dbMap*>; //vector of rows
    while (sqlite3_step(stmt) != SQLITE_DONE){
        stpRow = new dbMap; //map for one row

        for (int col = 0; col <
            sqlite3_column_count(stmt); col++) {
            const char* name =
                sqlite3_column_name(stmt, col);

            const unsigned char* val =
                sqlite3_column_text(stmt, col);
            string s1 = string(name);
            string s2 = string((const char*)val);

            stpRow->insert({s1, s2}); //add to map
        }
        //add that map to the row vector
        stpRows->push_back(stpRow);
    }
    rewindStatement(); //you can comment this out
    return new Results(stpRows); //create Results
}
```

If this looks quite similar to the similar code for the Query execute method we wrote above, it is. In fact, other than the fact that it avoids that static variable callback kludge, it is the same kind of code. The only difference is that you must finalize (delete) that statement yourself when you are done. The main difference is that we create the row map and the rows vector using **new** and access them as pointers.

```
void sqltPrepQuery::closeStatement(){
    sqlite3_finalize(stmt);
}
```

In fact, you can use the sqltPrepQuery all the time, without binding any variables to the query if there are none. You just

have to remember to finalize that statement so its memory is released.

As far as the **rewind** method is concerned, we included a call to it at the end of our query execute method. There is no real reason not to do this, as it simply resets a single pointer to the top of the compiled SQL code. But if you prefer, you can comment that statement out and call it yourself.

Our code for it just calls the internal reset method:

```
void sqltPrepQuery::rewindStatement() {
    sqlite3_reset(stmt);
}
```

Summary

We have seen some general ways to build classes for accessing databases. This strategy is called a *Facade pattern* and you can use it and much of the code we wrote to access nearly any other database, many more simply than what SQLite required.

Example programs on GitHub

1. Sqlitest – the simplest example for SQLite
2. Simple SQLite Database classes – all classes in one file
3. SQLite external classes – same as 2 but all classes and headers in separate files
4. SQLite Tables – Creates the grocery database using Table methods
5. SQLite Prepared – example code for prepared statements

References

1. SQLite Studio -- <https://sqlitestudio.pl/>
2. SQLite download -<https://sqlite.org/download.html>
3. When to use SQLite - <https://sqlite.org/whentouse.html>
4. The command line interface -- <https://sqlite.org/cli.html>

18. Using the MySQL database

MySQL is an open source full-featured database system that has all the power of other commercial databases. Unlike SQLite, MySQL can handle any number of different database projects in the same server environment. Each one is a completely separate set of tables. This leads to some terminology confusion as to what thing you refer to as a “database.”

MySQL solves this problem by referring to each individual database as a “Schema.” But, of course, the SQL itself still refers to opening each “database.” However, for internal consistency, MySQL also allows you to open a “Schema.” Let’s just decide that MySQL is a Database Manager package which contains any number of databases.

While MySQL started out, it was an open source project, but it was eventually sold to Sun Microsystems, which was then taken over by Oracle. Oracle now supports MYSQL for free although it offers a paid version as well. Some of the original MySQL developers left the project, taking the MySQL code with them, and forming the new open-source project MariaDB, which is also freely available.

Installing MySQL

For Windows, and most other platforms, you want to download the MySQL installer¹ and at the very least, install MySQL and the C++ Connector code. The installer window is shown in Figure 18-1.

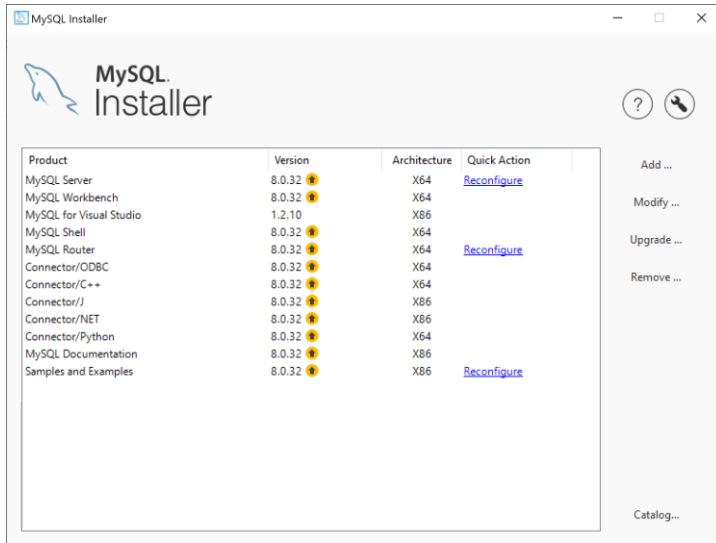


Figure 18-1 - MySQL Installer

You also want to install the MySQL Workbench, which just like the SQLite Studio, provides with a visual interface to inspect databases and run queries.

If you install the server and connector to the default windows directories, you will find the Connector C++ code installed under `c:\Program Files\MySQL\Connector C++ 8.0\`. The Connector C++ 8.0 library is a set of C++ classes you can use to connect to MySQL. They are at a considerably higher level than those in SQLite and at a lower level than the Python interface to MySQL we discussed previously.⁵

Writing C++ to connect to MySQL

All of the classes you need to connect to MySQL are in the **mysqlx** namespace, and are pretty straightforward to use. There are some examples in XDevAPI manual.⁶ But we'll show you how in this discussion.

In Visual Studio, you need to create a Console C++ project, and add the following to the Release Configuration Property Pages. Connector C++ 8.0 requires C++ version 11, but will work OK with C++ version 14, with only minor warning messages. However, your code will *not* compile correctly with more recent C++ versions. This is a real shame as it takes away the advantages of the last 10 years of C++ improvements.

- Be sure that the C++ language standard is set to C++ 14.
- Set VC++ External Includes to

```
c:\Program Files\MySQL\Connector C++ 8.0\include
```

- and set the VC++ library directories to

```
C:\Program Files\MySQL\Connector C++ 8.0\lib64
```

- You also need to add this path under Linker/General/Additional Library directories. And finally you need to add a PATH statement under Configuration Properties/Debugging. It should be of the form

```
PATH=C:\Program Files\MySQL\Connector C++ 8.0\lib64;%PATH%
```

- And finally, under Linker/Input edit the entry to start with

```
mysqlcppcon8.lib
```

- It is also important that that lib file be in the same folder as the DLL files. You should copy it there from the vs14 directory just below the lib64 directory.

Now we can write a little C++ program to connect to MySQL. First, you must log in to the database using port 33060.

Our program must start by including the header files for the Connector code, as well as the namespace `mysqlx`.

```
#include <iostream>
#include <string>
#include <mysqlx/xdevapi.h>

using std::cout;
using std::endl;
using namespace::mysqlx;
```

Note that to connect to the older interface in Python we used for 3306. For Connector C++ we use port 33060.

```
try {
    std::string user = "newuser";
    std::string password = "new_user";
    std::string host = "localhost";
    unsigned int port = 33060;

    // Create a session to connect to the MySQL Server
    mysqlx::Session session(host, port,
                            user, password)
```

Then, we select a database using the SQL statement USE. We'll assume the groceries database is already there, but you can use any database you want. By the end of this chapter, we'll have written much the same code as in the last chapter to create that groceries database. In any case we select the database here:

```
// Tell MySQL which database to use
session.sql("USE groceries").execute();
```

Next we write a little query:

```
// Perform any operations on the table or query data
string sql = "Select distinct foodname, price, storename
from prices \
    join foods on(foods.foodkey = prices.foodkey) \
    join stores on(stores.storekey = prices.storekey)";
```

and then we execute it:

```
//execute a query and print out the result
SqlResult res = session.sql(sql).execute(); //query result
```

And finally, we get the column names array:

```
//array of columns
const Columns& columns = res.getColumns();
```

and run through the rows, printing out the column values:

```
Row row;
while ((row = res.fetchOne())) {    //get each row
    //get each column
    for (unsigned index = 0;
         index < res.getColumnCount();
         index++)
    {
        //print out the column name and content
        cout << columns[index].getColumnName() <<
            ": " << row[index] << endl;
    }
}
```

Finally, we close the session and catch any exceptions:

```
// Close the session
    session.close();
}
catch (const mysqlx::Error& error) {
    std::cerr << "Error: " << error << std::endl;
    return 1;
}
```

Note that you must enclose all of your access to MySQL in a try-catch exception which will print out an error message if something fails, and that you must close the session. The output is much as before:

```
foodname: Apples
price: 0.27
storename: Stop and Shop
foodname: Oranges
price: 0.36
storename: Stop and Shop
```

```

foodname: Hamburger
price: 1.98
storename: Stop and Shop
foodname: Butter
price: 2.39

```

and so forth.

Debugging libraries for Connector C++

You can also download the debugging libraries so you can run your code in Debug mode. Go to this page⁴, scroll down and download the Windows 64-bit Zip archive Debug Libraries. You only need the DLLs, so unzip this archive and copy the **debug** folder under the **lib64** folder.

- Copy that debug folder under the lib64 folder of your Connector C++ installation.
- Create Visual Studio Console C++ project and under project Project / Properties set all the Release 64 properties as above.
- Then switch the Properties drop-down to Debug 64 and set VC++ Library Directory to

```
C:\Program Files\MySQL\Connector C++ 8.0\lib64\debug
```

- Set the Linker /General/ Additional Library Directories to the same thing.
- Set the Configuration Properties/Debugging /Environment to start with

```
PATH=C:\Program Files\MySQL\
Connector C++ 8.0\lib64\debug;%PATH%
```

- Copy the mysqlcppcon8.lib file into that same ...lib64\debug folder.

Creating C++ classes to connect to MySQL

Now we are going to create versions of the same classes we used in the SQLite chapter. As we noted there, this is effectively a

Facade pattern, where we wrap the code to communicate with MySQL in simpler class structure. And since we are already dealing with a class based interface in Connector C++, this is very easy indeed.

We'll work from an abstract Database class which is the base class we used in the SQLite code as well.

```
#include "globaldefs.h"
class Database {
public:
    virtual int open(std::string filename) = 0;
    virtual int create(std::string filename,
                      bool del) = 0;
    virtual string getName() =0;
    virtual vector<string> getTableNames() =0;
    //close the database connection
    virtual void close() = 0;
};
```

Then to communicate with MySQL we need only fill in these simple methods in a derived msqDatabase class. Our class constructor opens a database session with the hostname, username, password and port:

```
// Create a session to connect to the MySQL Server
msqDatabase::msqDatabase(string host, int port,
                        string user, string password) {
    session = new Session(host, port, user, password);
}

//get a pointer to the current session
mysqlx::Session* msqDatabase::getSession() {
    return session;
}
```

In a similar fashion, we can select a database (or schema) by running a simple SQL query using the session.sql.execute method. Here's how we open a database:

```

// Create a session to connect to the MySQL Server
msqDatabase::msqDatabase(string host, int port,
    string user, string password) {
    session = new Session(host, port, user, password);
}

//get a pointer to the current session
mysqlx::Session* msqDatabase::getSession() {
    return session;
}

```

And, creating a database is just about as simple:

```

//create new database, delete prior vsn if del is true
int msqDatabase::create(string name, bool del) {
    dbName = name;
    if (del) {
        session->sql("DROP DATABASE IF EXISTS "+
            dbName).execute();
    }
    session->sql("create database " + dbName).execute();
    setSchema();
    return 0;
}

```

We also include two methods you can use to run SQL commands: one that returns results and one that doesn't.

```

//runs any sql that does not produce a Result
void msqDatabase::runSql(string sqltext) {
    session->sql(sqltext).execute();
}

//run any sql that returns an SqlResult
mysqlx::SqlResult msqDatabase::runQuery(string sqltext) {
    return session->sql(sqltext).execute();
}

```

In theory, the above is all you need to execute all queries in MySQL, but returning the query results in convenient maps is a worthwhile addition, and we will see that this takes very little

new code. There are only slight modifications to the Query class code and essentially none to the Results class code.

Numeric types in MySQL

MySQL has a greater range of numeric types compare to SQLite. They are shown in Table 18-1 and Table 18-2.

Name	Bytes
TINYINT	1
SMALLINT	2
MEDIUMINT	3
INT	4
BIGINT	8

Table 18-1 - Integer types

Name	Bytes
FLOAT	4
DOUBLE	8

Table 18-2 - Floating point types

MySQL also supports the DECIMAL type, for use with currency, where exact values are required.

MySQL Query results

Unlike SQLite, MySQL returns numeric values rather than strings in query results. The return value is an `mysqlx::Value` object which contains a data type and a data value. The types Value contains are shown in Table 18-3.

VNULL	null
UINT64	Unsigned integer
INT64	Signed integer
FLOAT	32 bit float
DOUBLE	64 bit double
BOOL	Boolean
DOCUMENT	Document
STRING	String
RAW	Raw bytes

ARRAY	Array of values
-------	-----------------

Table 18-3 - Types of data in Value object

So, the query results are mapped to a new dbMap which contains that Value object:

```
typedef map<string, mysqlx::Value> dbMap;
```

The actual Query code that constructs that map is quite similar to that we wrote for SQLite:

```
//execute the query and get back the col names and values
Results* Query::execute() {
    stpRows = new vector<dbMap*>; //vector of rows
    //run the query
    mysqlx::SqlResult res = db->runQuery(sql);
    //get the columns
    const mysqlx::Columns& columns = res.getColumns();
    mysqlx::Row row;
    while ((row = res.fetchOne())) {
        //create a map for this row
        dbMap* stpRow = new dbMap;
        for (unsigned index = 0;
            index < res.getColumnCount();
            index++)
            {string s1 =
                columns[index].getColumnName();
                //add to current map

                stpRow->insert({ s1, row[index]});
            }
        //add complete row to vector
        stpRows->push_back(stpRow);
    }
    //return Results object
    return new Results(stpRows);
}
```

Why does printing out a Value object work?

The printing of the results from a query amounts to

```
cout << dm["foodname"] << " " << dm["price"] << " "
      << dm["storename"] << endl;
```

If `dm["price"]` is a Value object instead of a number or string, why does this `cout` call work? It works because the Value object *overloads* the `<<` operator, and decides internally what data is to be sent to `cout` to be printed. If you want to get the value of one of these fields as a string, to put in a visual table, you have a couple of options.

One way is to “print” the contents of the Value object to a **stringstream** instead of to the console stream:

```
mysqlx::Value v = dm1["price"];
stringstream ss;           //create a stringstream
ss << v;                   //print the value to the stream
string outs = ss.str();   //convert to a string
```

Another simple way would be to check the type of the Value and then convert it to a string yourself using ifs or a **switch** statement:

```
string sval = "";           //answer goes here
int itype = r.getType();   //get the type
if (itype == r.FLOAT) {   //if float
    float val = float(r); //cast it
    sval = std::to_string(val); //convert it
}
else if (itype == r.INT64) {
    int val = int(r);     //here it's an integer
    sval = std::to_string(val);
}
else if (itype == r.STRING) {
    string val = string(r);
    //if a string already do nothing
    sval = val;
}
```

Creating the MySQL groceries database

Everything else is much the same as in SQLite. You can create and load the tables in just the same way. Here is a bit of the code:

```
try {
    std::string user = "newuser";
    std::string password = "new_user";
```

```

std::string host = "localhost";
unsigned int port = 33060;

// Create a session to connect to the MySQL Server
msqDatabase* db =
    new msqDatabase(host, port, user, password);
db->create("mygroceries");

//make the food table
msqTable* foodTable = new msqTable("foods", db);
foodTable->addColumn(
    new PrimaryCol("foodkey", true));
foodTable->addColumn(new CharCol("foodname", 45));
foodTable->create();

```

Prepared Queries in MySQL

You can carry out prepared queries using the C++ Connector 8.0 as well, although they differ slightly from those provided in SQLite. As in the prior case, you create a compiled SQL statement by leaving off the trailing `.execute()` method.

```
mysqlx::SqlStatement sqQuery= session->sql(sqltext);
```

So, that `sqQuery` object can have variable bound to it. However, if you look at the Connector C++ documentation, this object is not defined. It is only mentioned as the output of the `sql` method. For whatever reason, that `SqlStatement` object is not a first class object. For example, it has no default constructor, so you cannot write a class containing a `SqlStatement` variable. Writing code such as

```
mysqlx::SqlStatement sqQuery; //define a sqQuery instance variable
```

will be flagged by the compiler as an error, because it cannot create an object with no default constructor. There is a simple way around this, however. Let's create a method in our `msqDatabase` class:

```
//return an SqlStatement object
```

```
mysqlx::SqlStatement
    msqDatabase::getSqlStatement(string sqlp) {
    return session->sql(sqlp);
}
```

We can call this method from our **main** routine where you can create an use such an obkect:

```
mysqlx::SqlStatement sqQuery = db->getSqlStatement(sqlp);
```

Then we can pass a pointer to this object into our new **PreparedStatement** class as by simply writing

```
msqPrearedQuery* prep =
    new msqPreparedStatement(db, &sqQuery);
```

Finally, we can bind variables to that prepared statement.

Let's write the full example now. Our prepared query has two variables to be filled in, one for a price and one for a name. We set then using the same sort of **setVariable** method we used in SQLite, but note that there is no variable number. They are applied in the order they appear in the query.

```
//using prepared query
    string sqlp =
    "Select distinct foodname, price, storename from prices \
    join foods on(foods.foodkey = prices.foodkey) \
    join stores on(stores.storekey = prices.storekey) \
    where price > ? and storename like ?";

// create the statement
mysqlx::SqlStatement sqQuery =
    db->getSqlStatement(sqlp);
msqPrearedQuery* prep =
    new msqPreparedStatement(db, &sqQuery);

//and bind the values to the variables
prep->setVariable(2.0);
prep->setVariable("S%");
Results* res1 = prep->execute();
```

The code in the `PreparedStatement` class is very much the same as in the SQLite version, except that the `setVariable` methods do not include an index value: they are taken in order.

```
msqPreparedStatement::msqPreparedStatement(msqDatabase* pdb,
mysqlx::SqlStatement* psQuery):Query(pdb, "") {
    sqQuery = psQuery;
    db = pdb;
}

//bind a text value to the query
void msqPreparedStatement::setVariable(string value) {
    sqQuery->bind(value);
}

//bind a double value to the query
void msqPreparedStatement::setVariable(double value) {
    sqQuery->bind(value);
}
```

And, finally the `execute()` method is very much like the one in our `Query` class, except that we run the query that we have just bound the values to. It starts out:

```
//execute the query
//and get back the column names and values
Results* msqPreparedStatement::execute() {
    stpRows = new vector<dbMap*>; //vector of rows
    //run the bound query
    mysqlx::SqlResult res = sqQuery->execute();
```

Table functions in Connector C++

Connector C++ provides a convenient syntax for adding rows to a table. For our `foods` table, you could add a new food like this:

```
Schema schema = session.getSchema("mygroceries");
session.sql("USE groceries").execute();

Table table = schema.getTable("foods");
table.insert("foodname").values("Pineapple").execute();
```


Note that you can add strings to a VARCHAR without including those annoying extra escaped quotes. You can also add mixed types without any special syntax:

```
Table tb2 = schema.getTable("prices");
tb2.insert("storekey", "foodkey", "price")
        .values(3, 8, 3.75).execute();
```

Other approaches to prepared queries

In addition to relational tables, MySQL supports Collections. Collections contain documents in JSON format, but are stored in a more efficient binary format

According to the MySQL documentation, MySQL caches and remembers the last query and keeps it in compiled form unless you issue a new one. While this feature is described in the documentation for use with Collections, it also works with relational tables as we illustrate below.

This SQL fragment represents a query with a couple of variable values in it just as we used above:

```
where price > ? and storename like ? ";
```

Then you can bind two values to these variables in a single statement:

```
SqlResult res = session.sql(sql).
        bind(1.0).bind("S%").execute();
```

and after storing that result, you can simply bind new values to the same SQL, and they will be bound to that already compiled version of the query:

```
res = session.sql(sql).bind(0.5).bind("V%").execute();
```

If you insert some timing code around these two queries, you will find that the second one runs faster.

```
clock_t start, end;      //two clock objects
double cputime;         //time goes here
```

```

cout << std::fixed;           //fixed precision output
start = clock();             //start the clock
SqlResult res = session.sql(sql).bind(1.0).
    bind("S%").execute();
end = clock();               //end the clock
cputime = ((double)end - start) ; //save the time
cout << std::setprecision(9) << cputime << endl;

```

While this can be useful it is difficult to use this feature in classes as we have done previously, but it may be useful to you in other contexts.

Example programs on GitHub

1. MySQL test app – Opens the database and runs a simple query.
2. MySQL db tables – Using a complete C++ set of classes, it creates the complete groceries database and runs a query on it.
3. MySQL Prepared query – Carries out Prepared queries on the groceries database.
4. MySQL table rows – Uses new syntax to add rows to table.
5. Timing prepared – times prepared statements

Summary

In these two database chapters, we've covered a local database and a server database, and developed a Façade that allows you to use the same code in both systems. We've made some powerful use of the **map** object to return query results and tuples to create ways to add data to tables.

References

1. MySQL installer:
<https://dev.mysql.com/downloads/installer/>

2. MySQL Workbench:
<https://dev.mysql.com/downloads/workbench/>
3. MySQL Connector C++:
<https://dev.mysql.com/downloads/connector/cpp/>
4. MySQL Debug Binaries:
<https://dev.mysql.com/downloads/connector/cpp/>
5. Cooper, James W. *Python Programming with Design Patterns*, Pearson Education: New York:2022.
6. Working with XDevAPI: <https://dev.mysql.com/doc/x-devapi-userguide/en/devapi-users-introduction.html>
7. The `mysqlx::SqlStatement`,:
<https://dev.mysql.com/doc/dev/connector-python/8.0/mysqlx.SqlStatement.html>
8. Working with Prepared Statements:
<https://dev.mysql.com/doc/x-devapi-userguide/en/working-with-prepared-statements.html>

19. Namespaces and Modules

Namespaces are a unique part of C++ that allow you to have similar class and function names in each namespace. The closest thing to these in Python are folders which can each contain similarly named classes and methods.

But as we have seen, most libraries have their own namespaces. The largest, of course, is the **std** namespace, which has many thousands of entries. But most of the libraries we have dealt with, such as **wxWidgets** and **armadillo** also have their own namespaces that you access with the **using** directive. As your projects get larger, you begin to see the advantages of dividing your code into different namespaces, and in C++ this is very easy indeed.

Let's create a little math routine namespace, and call it **minimath**. It might have a simple **square** function such as:

```
double square(const double x) {
    return x * x;
}
```

To put it in its own namespace, you simply surround the functions with a **namespace** declaration enclosed in braces:

```
namespace minimath {
    double square(const double x) {
        return x * x;
    }
}
```

Then, you can refer to it in your main program with the **using** directive:

```
using minimath::square;

int main() {
    double y = 123.45;
    double z = square(y);
    cout << "x^2 =" << z << endl;
```

```
}
```

But, this will only compile and run if you tell your main program that that namespace exists somewhere, by using an include file as usual. Here is the file `minimath.h`:

```
namespace minimath {
    double square(const double x);
}
```

which just declares that the **square** function exists in the **minimath** namespace. So your complete **main** program needs to refer to that include file and any others you use as usual.

```
#include <iostream>
#include "minimath.h"

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

using minimath::square;

int main() {
    double y = 123.45;
    double z = square(y);
    cout << "x^2 =" << z << endl;
}
```

Of course, your math library might contain more than one function, and they can be in the same or separate files. Here's a separate file for a cube function:

```
namespace minimath {
    double cube(const double x) {
        return x * x * x;
    }
}
```

You can use the same include file for both members of the same name space: it just has one more entry:

```
namespace minimath {
    double square(const double x);
```

```

        double cube(const double x);
    }

```

With that one change, our main program can then be:

```

using minimath::square;
using minimath::cube;

int main() {
    double y = 123.45;
    double z = square(y);
    cout << "x^2 =" << z << endl;
    cout << "x cubed is : " << cube(y) << endl;
}

```

And that is really about all there is to using namespaces.

Modules

Modules are groups of classes and functions that are compiled separately and then imported into your program. This makes compiling your own program much faster. The whole idea of modules is hardly new, but they didn't really come into their own until C++ 20, and as this is written in 2023, very few compilers besides Microsoft Visual Studio support them very well.

Modules have a lot of advantages besides speeding up compilation. They do away with the need for a lot of confusing include files where the order of those includes can be critical to your success. In fact, you can compile the whole **std** library into a module and do away with a lot of those includes [3].

Let's start with a really simple module containing those two math functions we wrote above. To create a module, you start with the **module** declaration, and follow with the code for that module. We'll call this module **Minimath** (note the capital letter for this one to distinguish it from the namespace we just discussed). Our simple module could then be

```

module Minimath;

double square(const double x) {

```

```

    return x * * x;
}

```

And, of course you can have several files containing additional functions (or classes) by simply preceding them with that same module declaration. Here is the cube function:

```

module Minimath;

double cube(const double x) {
    return x * * x * * x;
}

```

The module descriptor file

To make a Visual Studio project that contains modules, you must create a module definition file. In Visual Studio, these have an **.ixx** extension. Thus far, this extension is not standardized by other compilers, but the contents are. To create this definition file select Project/Add module... This brings up a window where you can create that file as shown in Figure 19-1. Be sure to type in the name of the module in the Name field at the bottom.

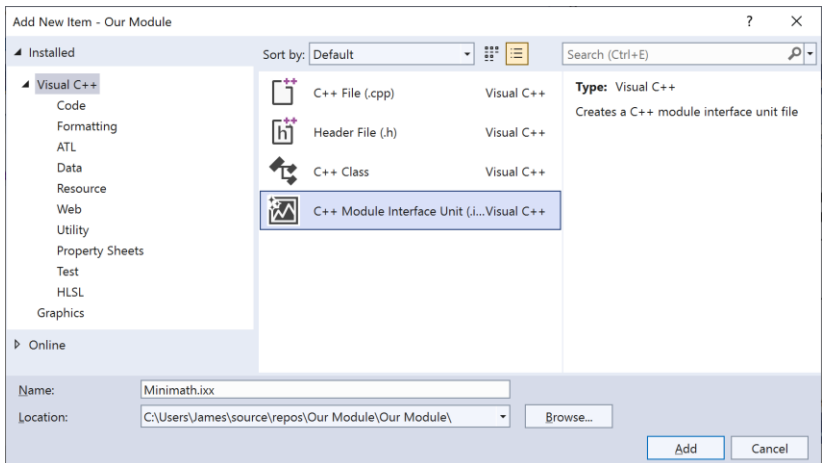


Figure 19-1 - Creating module definition file

This will create a file `Minimath.ixx` with the following temporary contents:

```
export module Minimath;
export void MyFunc();
```

Change the file to contain the actual function names:

```
//module descriptor file
export module Minimath;
    export double square(const double x);
    export double cube(const double x);
```

Then, create the main calling program that imports that module:

```
#include <iostream>
using std::cout;
using std::endl;

import Minimath;    //import the Minimath module

int main() {
    double x = 12;
    cout << "x^2 =" << square(x) << endl;
    cout << "x cubed is" << cube(x) << endl;
}
```

Make the descriptor file an include file

Before you can compile and run this simple example, you have to make one more change. You must make the module descriptor file an include file. To do this, right-click on the Headers line in the Solution Explorer window, and select **Add...** and then **Existing item**, and select the `Minimath.ixx` file. Once you have done that, the program should compile and run.

Combining namespaces and modules

Of course, if you are making actual, useful modules, you probably would put them in their own namespace.

In that case, the namespace declaration goes under the module declaration like this:

```
module Minimath;
namespace minimath {
    double square(const double x) {
        return x * x;
    }
}
```

and here:

```
module Minimath;
namespace minimath {
    double cube(const double x) {
        return x * x * x;
    }
}
```

The module descriptor file contains both:

```
export module Minimath;
namespace minimath {
    export double square(const double x);
    export double cube(const double x);
}
```

And the main calling program needs both the **import** and the **using** declarations:

```
import Minimath;
using minimath::square;
using minimath::cube;

int main() {
    double y = 123.45;
    double z = square(y);
    cout << "x^2 =" << z << endl;
    cout << "x cubed is : " << cube(y) << endl;
}
```

Example code on GitHub

1. Tinynamespace – namespace example
2. Our module – Minimath module example
3. Tinymodule – combines namespace and module code

References

1. Microsoft examples of namespaces.
<https://learn.microsoft.com/en-us/cpp/cpp/namespaces-cpp?view=msvc-170>
2. Microsoft explanation of modules:
<https://learn.microsoft.com/en-us/cpp/cpp/tutorial-named-modules-cpp?view=msvc-170>
3. Making the whole standard library into a module:
<https://learn.microsoft.com/en-us/cpp/cpp/tutorial-import-stl-named-module?view=msvc-170>

Part III-Design Patterns

Design patterns are a set of algorithms that computer scientists discovered when studying well-written computer code. They are not “written” so much as “curated” from examples of good object-oriented coding practices. In other words

Design patterns are frequently used algorithms that describe convenient ways for classes to communicate.

Design patterns began to be recognized more formally in the early 1990s by Erich Gamma¹, who described patterns incorporated in the GUI application framework, ET++. The culmination of these discussions and a number of technical meetings was the publication of *Design Patterns -- Elements of Reusable Software*, by Gamma, Helm, Johnson and Vlissides.² This book, commonly referred to as the Gang of Four or “GoF” book, has had a powerful impact on those seeking to understand how to use design patterns and has become an all-time best seller. It describes 23 commonly occurring and generally useful patterns and comments on how and when you might apply them. We will refer to this groundbreaking book as *Design Patterns*, throughout this book.

Since the publication of the original *Design Patterns* text, there have been a number of other useful books published. These include our popular *Python Programming with Design Patterns*,³ and an analogous book on Java Design Patterns.⁴

Notes on Object Oriented Approaches

The fundamental reason for using design patterns is to keep classes separated and prevent them from having to know too much about one another. Equally important, using these patterns helps you avoid reinventing the wheel and allows you to describe your programming approach succinctly in terms other programmers can easily understand.

There are a number of strategies that OO programmers use to achieve this separation, among them encapsulation and inheritance. Nearly all languages that have OO capabilities support inheritance. A class that inherits from a parent class has access to all of the methods of that parent class. It also has access to all of its variables. However, by starting your inheritance hierarchy with a complete, working class you may be unduly restricting yourself as well as carrying along specific method implementation baggage. Instead, *Design Patterns* suggests that you always

Program to an interface and not to an implementation.

Putting this more succinctly, you should define the top of any class hierarchy with an *abstract* class or an *interface*, which implement no methods, but simply define the methods that class will support. Then, in all of your derived classes you have more freedom to implement these methods as most suit your purposes.

The other major concept you should recognize is that of *object composition*. We have already seen this approach in our Statelist examples. This is simply the construction of objects that contain others: encapsulation of several objects inside another one. While many beginning OO programmers use inheritance to solve every problem, as you begin to write more elaborate programs, the merits of object composition become apparent. Your new object can have the interface that is best for what you want to accomplish without having all the methods of the parent classes. Thus, the second major precept suggested by *Design Patterns* is

Favor object composition over inheritance.

At first this seems contrary to the customs of OO programming, but you will see any number of cases among the design patterns where we find that inclusion of one or more objects inside another is the preferred method.

Commonly used Patterns

We've already seen how to use the Command Pattern and the Mediator Pattern in building our first user interfaces in Chapter 13. And we learned about the Façade Pattern when we wrote interfaces to two databases in Chapter 17 and 18. In the chapters that follow, we'll look at Factory Patterns, Adapter patterns and Bridge patterns, which are some of the most commonly used patterns beside the Command and Mediator patterns.

References

1. Erich Gamma, *Object-Oriented Software Development based on ET++*. (in German) Springer-Verlag, Berlin, 1992.
2. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA: 1995.
3. James Cooper, *Python Programming with Design Patterns*, Pearson Education, 2022.
4. James Cooper, *Java Design Patterns: A Tutorial*, Addison-Wesley: Boston: 2000

20. Factory Patterns

Some of the simplest of the Design Patterns are the Factory Patterns: programs where the code itself decides which of several related classes to use. Note that the user does not make these decisions, the Factory itself decides which class to instantiate.

The Simple Factory Pattern

We'll start with the Simple Factory Pattern, which we use in an elementary program to decide whether a name is entered

- lastname, firstname or
- firstname lastname

We'll start by creating a base Namer class that holds the first name and the last name, and provides accessor methods for fetching them:

```
//base namer class
class Namer {
protected:
    string fname{ NULL }; //first name
    string lname{ NULL }; //last name

public:
    string getFname() {
        return fname;    //get the first name
    }
    string getLname() {
        return lname;    //get the last name
    }
};
```

Then, we can create two derived classes, one for first-last and one for last-first. The underlying assumption in this simple program is that first-last name entries are separated by a space, and last-first separated by a comma. Both of these classes are

derived from `Namer` and store their data in `fname` and `lname` inside that base class.

So, the `FirstLast` class just looks for the space and cuts the string into two pieces on either side of that space.

```
//splits apart two names separated by a space
class FirstFirst : public Namer {
public:
    FirstFirst(string nameEntry) {
        //find the space
        int index = nameEntry.find(" ");
        if (index > 0) {
            //if we find one split there
            fname = nameEntry.substr(0, index);
            lname = nameEntry.substr(index+1,
                size(nameEntry) - index);
        }
    }
};
```

And, similarly, the `LastFirst` splits at the comma,

```
//splits apart names separated by a comma
class LastFirst : public Namer {
public:
    LastFirst(string nameEntry) {
        int index = nameEntry.find(",");
        if (index > 0) {
            lname = nameEntry.substr(0, index);
            //prevents error if no 2nd name
            if (index+2 < size(nameEntry))
                fname = nameEntry.substr(index+2,
                    size(nameEntry) - (index+2));
        }
    }
};
```

The `Factory` itself simply looks for that comma and returns a pointer to an instance of `LastFirst`. Otherwise, it returns a pointer to an instance of `FirstFirst`.

```

//The name factory returns a pointer
// to a last-first namer
// or a first-first namer
class NameFactory {
private:
    string nameString;
public:

    NameFactory(string nm) {
        nameString = nm;           //save the string
    }
    //get the right child of the Namer class
    Namer* getNamer() {
        //if there is a comma
        int index = nameString.find(",");
        if (index > 0) {
            return new LastFirst(nameString);
            //return a LastFirst instance
        }
        //otherwise, return a FirstLast instance
        else {
            return new FirstFirst(nameString);
        }
    }
};

```

The calling program then reads in a line from the console and uses the NameFactory to decide which Namer class to use:

```

//Program to take in names with spaces or commas
//and determine which is last name and which is first
int main() {
    bool quit = false; //quit if it becomes true
    char name[100];     //buffer where the line is read
    while (!quit) {    //repeat until "quit" entered
        //get the name chars from the keyboard
        cout << "Enter name: ";
        cin.getline(name, 100);
        string nm(name); //convert to a string
        if (nm == "quit") {
            quit = true; //set flag for "quit"
        }
        else {

```

```

NameFactory nf(nm); //create a name factory
//get the right namer pointer

Namer* nmr = nf.getNamer();
//print out the first and last names

cout << format("First: {} Last: {} \n",
    nmr->getFrname(), nmr->getLname() );
delete nmr;    //delete pointer to class
    }
}
}

```

We can run the above code just as you see it here, or you can wrap it in a simple user interface based on our original Add two numbers example. The result is shown in Figure 20-20-1.

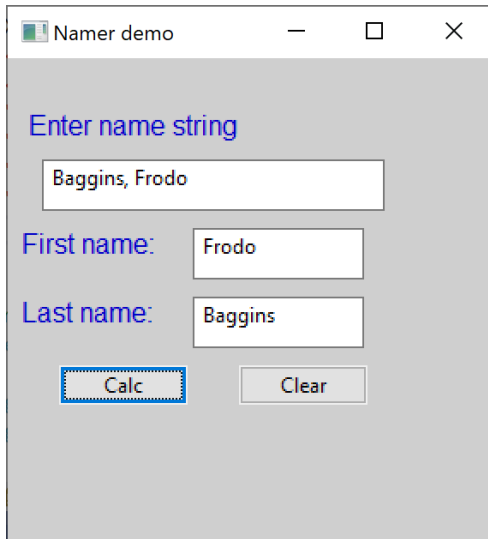


Figure 20-20-1 - The visual Namer demo

The Factory Method Pattern

The Factory Method Pattern is similar to the Factory Pattern, except that the decision as to which computation class to use is made by a derived class, and each derived class may choose a

different computation class. This is actually simpler than it sounds.

In both swimming and track and field races, the order in which athletes compete is determined not only by their entry time, but by whether the race will be run only once or as preliminaries and finals. In the first case, called Timed Finals, the athletes are arranged in heats so that the final heat contains the fastest athletes by their entry (or seed) time. In swimming, when the race is run as prelims and finals, the athletes are mixed together so that the fastest athletes are in one of the top 3 heats, giving more swimmers a chance to compete against the very fastest swimmers. Track and field races have a different, but analogous system.

We start by creating a Swimmer class, which holds the data for one swimmer's name, age and club and seed time. It also parses one line of the input files 100free.txt or 500free.txt.

```
class Swimmer {
private:
    string fname;           //first name
    string lname;          //last name
    int age;                //age
    string club;           //club symbol
    string seedtime;       //time with colon in string
form
    float time = 0.0;      //time as float for sorting
    int _lane = 0;         //private lane value
    int _heat = 0;         //private heat value

public:
    //remove colon to make float
    string removeColon();
    Swimmer(string dataline);
    Swimmer();
    string getName();      //Swimmer name
    string getClub();      //Swimmer club
    int getAge();          //swimmer age
    float getTime();       //get the value of seed time
    string getSeed();      //get the seed time string
};
```

```

int heat();           //get the heat number
void heat(int h);    //set the heat number
int lane();          //get the lane number
void lane(int ln);   //set the lane number
};

```

While the Swimmer class could be represented in any language, including Python, a C++ representation keeps the internal data private and uses getter methods to fetch the name, age, club and seed time. But since a program to place swimmers in heats and lanes require both read and write access to the heat and lane variables, we *could* make the public as we would do in Python. Or we could take advantage of polymorphism and create two versions of the **heat** and **lane** methods: one for fetching values and one for storing values as shown in the above header file. The actual code for those methods fetches and store values in the private variable which we name **_heat** and **_lane** so that we can use the original names in the access methods:

```

int Swimmer::heat() {
    return _heat;           //get the heat number
}
void Swimmer::heat(int h) {
    _heat = h;             //store the heat number
}
int Swimmer::lane() {
    return _lane;          //get the lane number
}
void Swimmer::lane(int ln) {
    _lane = ln;           //store the lane number
}

```

In C++ you can have several methods with the same name as long as the arguments are different. So, we can write

```

int mylane = sw->lane();
//or
sw->lane(mylane);

```

This makes the code less cluttered and easier to read. Note that this approach would not work in Python where this sort of polymorphism is not supported.

In our seeding factory method program, we then create an abstract Event class, which contains an array of swimmers and returns the seeding method it uses:

```
class Event {
protected:
    int numLanes;
    vector <Swimmer*> swimmers;
public:
    Event(string filename, int lanes);
    Event();
    virtual Seeding* getSeeding() = 0;
};
```

We derive from that has a TimedFinalEvent and a PrelimEvent.

We also create a base class Seeding, from which we derive StraightSeeding and CircleSeeding. StraightSeeding places the fastest swimmers in the first heat, the next fastest in the next heat and so forth. Circle Seeding places the fastest 3 swimmers in the middle lane of the first 3 heats and the next 3 fastest in the adjacent lane and so forth.

So, if we choose a TimedFinalEvent

```
TimedFinalEvent::TimedFinalEvent(string filename, int
lanes)
    : Event(filename, lanes){
}

Seeding* TimedFinalEvent::getSeeding() {
    return new StraightSeeding(swimmers, numLanes);
}
```

It chooses StraightSeeding and if we pick a PrelimEvent it chooses CircleSeeding.

```

PrelimEvent::PrelimEvent(string filename, int lanes)
    :Event(filename, lanes) {}

Seeding* PrelimEvent::getSeeding() {
    return new CircleSeeding(swimmers, numLanes);
}

```

Thus, the class itself decides which seeding to use, and this is the way the Factory Method Pattern works. To illustrate with the console version, we can either enter 1 or 5 for 100 Free or 500 Free. For younger swimmers, the 500 free is generally swum as a Timed Final event, so the result is shown here:

```

enter '1' or '5' (0 to quit): 5
13 3 Emily Fenn           17 WRAT 459.54
13 4 Kathryn Miller       16 WYW  501.35
13 2 Melissa Sckolnik     17 WYW  501.58
13 5 Sarah Bowman         16 CDEV 502.44
13 1 Caitlin Klick        17 MBM  502.59
13 6 Caitlin Healey       16 MBM  503.62
12 3 Kim Richardson       17 WYW  504.32
12 4 Beth Malinowski      16 HAC  504.77
12 2 Patricia Finnerty    17 WYW  505.76
12 5 Carolyn Bowman       15 CDEV 505.79
12 1 Katie Martin         17 CDEV 506.78
12 6 Lauren Dudley        17 WYW  508.96
11 3 Lori Schwanhausser   15 WYW  510.82

```

After Heat 11, the remaining swimmers are straight seeded:

```

10 3 Courtlandt McKinlay  15 CDEV 515.65
10 4 Kelly Ottenbreit     16 HNHS 516.29
10 2 Ashley Orchard       17 PCSC 517.43
10 5 Jessica Lasalle      17 CDEV 518.03
10 1 Stephanie Nickse     17 WYW  518.04
10 6 Rachel Reichard      17 MBM  518.52
 9 3 Sarah Martin         16 CDEV 519.35
 9 4 Grace Turman         16 BRS  519.49
 9 2 Kristyn Sayball      16 ARAC 519.89

```

You can also do this more elegantly using a simple GUI interface as shown in Figure 20-2.

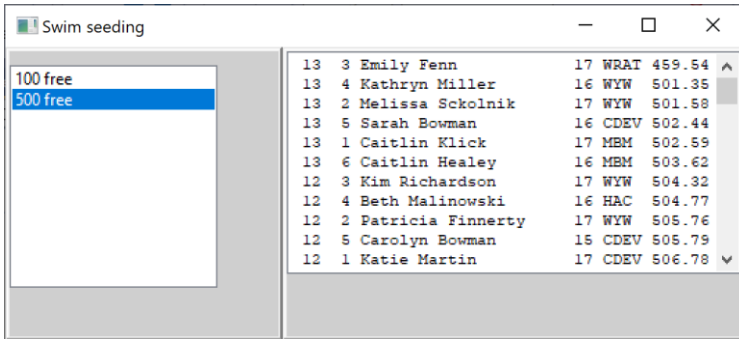


Figure 20-2 - GUI of straight seeding

For the characters to line up correctly, you have to change the font to a monospaced font such as Courier New. In wxWidgets, the method to use is

```
seedList->SetFont(wxFont(8, wxFONTFAMILY_SWISS,
wxFONTSTYLE_NORMAL,
wxFONTWEIGHT_NORMAL, false,
wxT("Courier New")));
```

Or you could use the Grid display widget instead as shown in Figure 20-3.

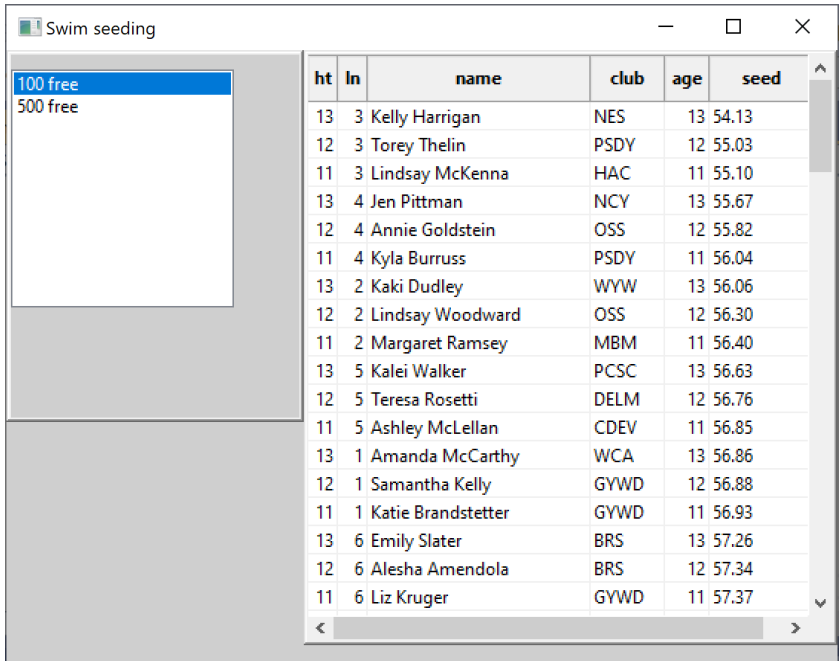


Figure 20-3 --Seeding GUI using the Grid widget

Example programs on GitHub

1. ConsoleNamer – console version of Simple factory.
2. UINamer - GUI version of Simple Factory.
3. SwimSeed – Console version of seeding factory in a single file
4. SwimSeedClasses – Console seeding divided in classes and headers.
5. SwimDisplay – GUI Seeding using 2 list boxes.
6. SwimGrid – GUI Seeding using the Grid widget.

21. The Abstract Factory Pattern

The Abstract Factory pattern is one level of abstraction higher than the factory pattern. You can use this pattern when you want to return one of several related classes of objects, each of which can return several different objects on request. In other words, the Abstract Factory is a factory object that returns one of several groups of classes. Which class from that group is to be used might even be decided by a Simple Factory.

One classic application of the abstract factory is the case where your system needs to support multiple “look-and-feel” user interfaces, such as Windows, Gnome or OS/X. You tell the factory that you want your program to look like Windows and it returns a GUI factory which returns Windows-like objects. Then when you request specific objects such as buttons, check boxes and windows, the GUI factory returns Windows instances of these visual interface components.

A GardenMaker Factory

Let’s consider a simple example where you might want to use the abstract factory in your application.

Suppose you are writing a program to plan the layout of gardens. These could be annual gardens, vegetable gardens or perennial gardens. However, no matter which kind of garden you are planning, you want to ask the same questions:

1. What are good border plants?
2. What are good center plants?
3. What plants do well in partial shade?

...and probably many other plant questions that we’ll omit in this simple example.

The Plant class

We'll start by defining a class for each plant. In this simple example, the Plant class stores only the plant's name and returns it as a string.

```
class Plant {
private:
    string plantName;           //the name
public:
    Plant(string pname) {       //constructor saves name
        plantName = pname;
    }
    string name() {            //returns the name
        return plantName;
    }
};
```

A Garden class

Then, we need to define a Garden class that contains several plants. Here, we create garden containing three types of plants:

```
//base Garden class
class Garden {
protected:
    Plant* sunnyPlant; //pointer to sunny plant
    Plant* shadePlant; //pointer to shade plant
    Plant* borderPlant; //pointer to border plant
    string name; //name of the Garden

public:
    Garden(const string gname) {
        name = gname; //save the name
    }
    string getName() {
        return name; //get the name
    }
    Plant* shade() { //get the shade plant
        return shadePlant;
    }
    Plant* sunny() { //get the sunny plant
        return sunnyPlant;
    }
};
```

```

    Plant* border() { //get the border plant
        return borderPlant;
    }
};

```

Now we create 3 classes derived from Garden for 3 types of gardens: Veggie, Perennial and Annual. Here are two of them:

```

class VeggieGarden :public Garden {
public:
    VeggieGarden():Garden("Vegetable") {
        sunnyPlant = new Plant("Corn");
        borderPlant = new Plant("Peas");
        shadePlant = new Plant("Broccoli");
    }
};

```

```

class AnnualGarden :public Garden {
public:
    AnnualGarden():Garden("Annual") {
        sunnyPlant = new Plant("Marigold");
        borderPlant = new Plant("Alyssum");
        shadePlant = new Plant("Coleus");
    }
};

```

So, matter what kind of Garden you choose, you can ask which plants are good as border, in full sun or in the shade. This is the crux of a Abstract Factory. You can see it in the visual interface in Figure 21-1.

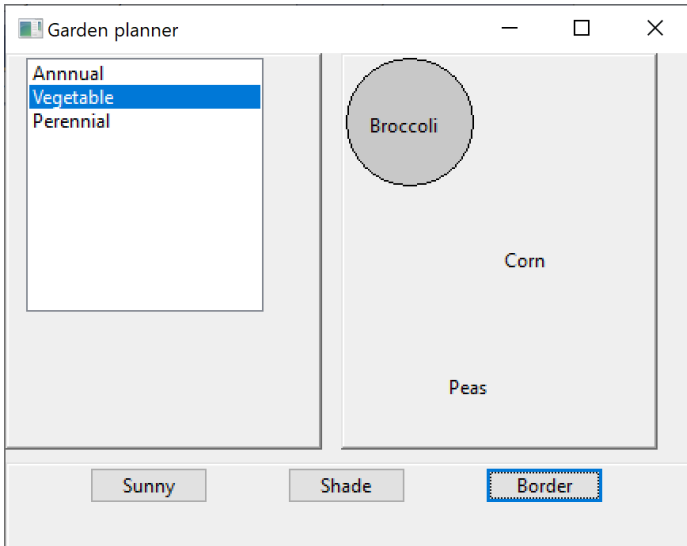


Figure 21-1- Garden planner abstract factory

How the GUI works

Whenever you have a GUI where there are several click events (listbox and 3 buttons), you probably should use a Mediator pattern to handle communication between the visual widgets. And this is particularly useful when you are working with systems like wxWidgets, where all the drawing takes place in a Paint event.

You must create a class derived from wxPanel that contains the paint event code:

```
//derived panel so we can create paint event
class PaintPanel : public wxPanel {
private:
    Mediator* med;
public:
    int width, height;
    PaintPanel(wxWindow* parent,
               wxWindowID id = wxID_ANY,
               const wxPoint& pos = wxDefaultPosition,
               const wxSize& size = wxDefaultSize,
               long style = wxTAB_TRAVERSAL,
```

```

        const wxString& name = wxPanelNameStr); {}
        //the paint event handler
        void OnPaint(wxPaintEvent& event);

        void setMediator(Mediator* med);
};

```

Note that **onPaint** is just a suggested name for the method that does the painting; you can call it anything you like. That onPaint method could look like this:

```

//PaintPanel is where the painting occurs
void PaintPanel::OnPaint(wxPaintEvent& event) {
    wxPaintDC dc(this);
    //fill color=gray
    dc.SetBrush(wxColor(200, 200, 200));
    //draw the shade circle
    dc.DrawCircle(wxPoint(40, 40), 40);

    //paint the names of the plants
    Garden* gd = med->getCurrentGarden();
    //coordinates obtained by trial and error
    dc.DrawText(wxString(gd->sunny()->name()),
        wxPoint(100, 120)); //sunny
    dc.DrawText(wxString(gd->shade()->name()),
        wxPoint(15, 35));
    dc.DrawText(wxString(gd->border()->name()),
        wxPoint(65, 200));
}

```

This is the simplest case, where all the plant names are drawn all the time. If we want to draw them only after the corresponding button has been clicked, you need to check the Mediator for a flag that says whether or not to draw each plant name yet.

```

if (med->showSun())
    dc.DrawText(wxString(gd->sunny()->name()),
        wxPoint(100, 120)); //sunny
if (med->showShade())
    dc.DrawText(wxString(gd->shade()->name()),
        wxPoint(15, 35));

```



```

if (med->showBorder())
    dc.DrawText(wxString(gd->border()->name()),
                wxPoint(65, 200));

```

The Mediator then has three variables representing whether to draw each of the plant names:

```

class Mediator {
private:
    vector<Garden*> gardens;
    wxListBox* listbox;
    wxFrame* frame;
    Garden* currentGarden;

    bool show_sun = false;
    bool show_shade = false;
    bool show_border = false;

```

And the three buttons in the display each set one of those variables to true. They are all reset to false when you select another garden type.

So we have here a fairly simple example of a Abstract Factory. The factory selects a garden which itself is made up of three Plant classes. In a more advanced system, the number of plants of each type could be variable.

Example program In GitHub

1. AbsFactory – Abstract Factory using the Garden Planner.

22. Adapters

The Adapter pattern is used to convert the programming interface of one class into that of another. We use adapters whenever we want unrelated classes to work together in a single program. The concept of an adapter is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface.

There are two ways to do this: by inheritance, and by object composition. In the first case, we derive a new class from the nonconforming one and add the methods we need to make the new derived class match the desired interface. The other way is to include the original class inside the new one and create the methods to translate calls within the new class. These two approaches, termed class adapters and object adapters are both fairly easy to implement in C++.

Moving Data between Lists

Let's consider a simple program that allows you to enter student names into a list, and then select some of those names to be transferred to another list. Our initial list consists of a class roster and the second list, those who will be doing advanced work, as shown in Figure 22-1

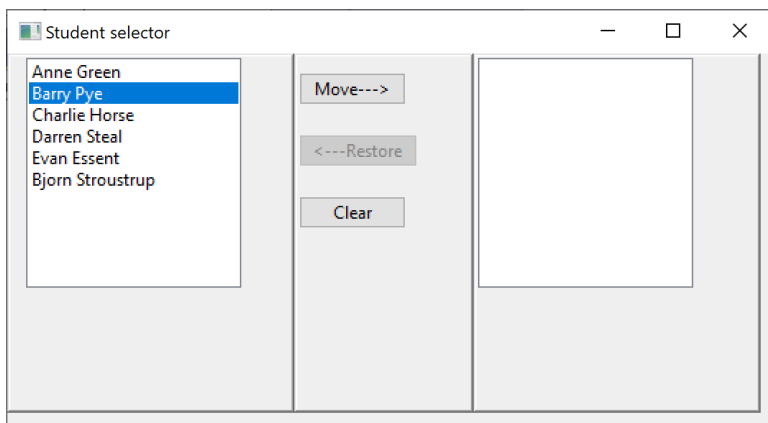


Figure 22-1 - Student selector- one selected

In this program you can select any name and the Move button becomes enabled. If you click on it, that student's name will be moved to the right-hand column of students selected for some special training, as shown in Figure 22-2. Note that the Move button is then disabled until you select another student.

The reverse also applies. If you select one student in the right-hand list, the Restore button is enabled, and you can move that student back into the left column.

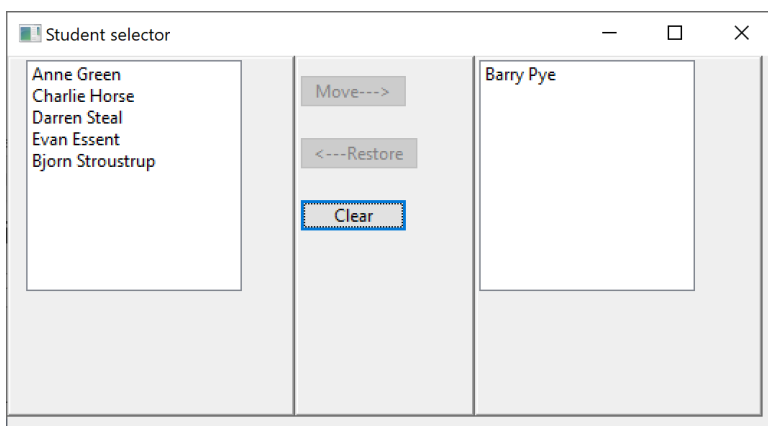


Figure 22-2 - Selected student moved to right column.

Now, since we are just moving names around, our students could just be strings. But, if you want to display the student data in a grid, you don't really have any data. In this simple example, we just display the lengths of the first and last names. This is shown in Figure 22-3.

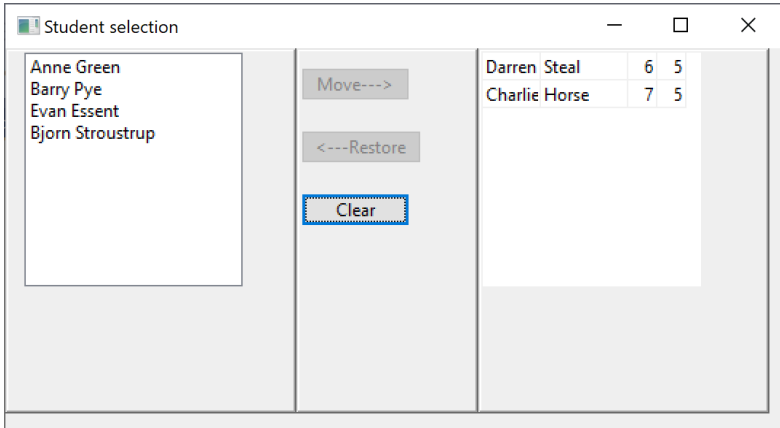


Figure 22-3 - Grid showing name lengths

So, how do we write an Adapter to deal with this simple problem. First, let's summarize the methods we created to manipulate the right hand list box in Figure 22-2. They are all methods of the `wxListBox`, and amount to

```
void Clear()
void Append()
void Delete(int index)
int GetSelection();
wxString GetString(int index);
```

Now, in `wxWidgets`, you have to connect click events (such as on listboxes) to your code. This is easiest if you use the `Bind` method in the same class the builds the GUI. The event connects to a method in that same class (because this is simplest) and then calls the Mediator to take care of the interactions between widgets. So, to connect the two listboxes to code we have a `Bind` call for each of them:

```

//connect the click events to the listboxes
studentList->Bind(wxEVT_LISTBOX,
&StudentFrame::onClick, this);
advList->Bind(wxEVT_LISTBOX,
&StudentFrame::advClick, this);

```

Then then two onClick methods call different methods in the Mediator:

```

//tells the Mediator the listbox has been clicked
void StudentFrame::onClick(wxCommandEvent& event) {
    med->onListClick(event);
}

//The right hand listbox has been clicked.
void StudentFrame::advClick(wxCommandEvent& event) {
    med->advListClick(event);
}

```

The Mediator can then handle the button enable/disable code, and note the selected name:

```

//left list box click comes here.
void Mediator::onListClick(wxCommandEvent& event) {
    moveButton->Enable();
    restoreButton->Disable();
}

//right list box click comes here.
void Mediator::advListClick(wxCommandEvent& event) {
    moveButton->Disable();
    restoreButton->Enable();
}

```

The Move and Restore button clicks fetch the name and move it to the other listbox:

```

//move name to right
void Mediator::moveClick() {
    int index = listBox->GetSelection();
    wxString nm = listBox->GetString(index);
    advListBox->Append(nm); //add name to right
}

```

```

listbox->Delete(index); //remove name from left
moveButton->Disable(); //disable until lb clicked
}

//move name to left
void Mediator::restoreClick() {
    int index = advListBox->GetSelection();
    wxString nm = advListBox->GetString(index);
    advListBox->Delete(index); //remove from right
    listbox->Append(nm); //add to left
    restoreButton->Disable(); //button disable
}

```

The Grid Adapter code

There are a couple of changes in connecting to the grid because of the event model that wxWidgets uses. First, you need to bind a different click event:

```

advGrid->Bind(wx.EVT_GRID_SELECT_CELL,
             &StudentFrame::advClick, this);

```

And in the Mediator, the advListClick method needs to pass the event to the Grid adapter, because it contains the row number that was clicked on, and there is no easy way to find out from the grid which cell was clicked on:

```

//right list box click comes here.
void Mediator::advListClick(wxGridEvent& event) {
    advListBox->onClick(event); //send the current row
    moveButton->Disable();
    restoreButton->Enable();
}

```

When you click on a cell of a grid row, it is helpful to highlight the entire row, as happens in a listbox. You can set that selection mode when you create the grid:

```

//select whole row
grid->SetSelectionMode(wxGrid::wxGridSelectRows);

```

This makes selection of a row look like Figure 22-4.

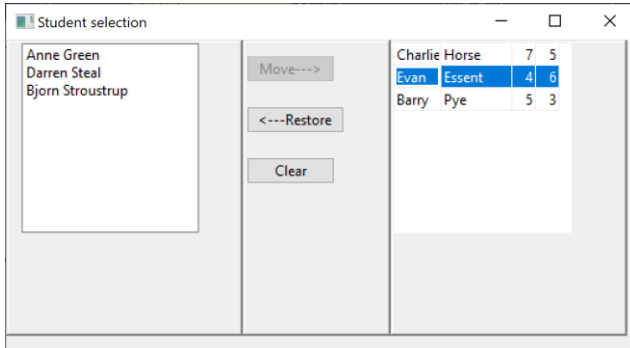


Figure 22-4- Shows selected row

But within the adapter code, the Append method is the only one that is radically different, because it splits each name into first and last name strings that then get placed in the first two columns. Then to show further information, it inserts the lengths of these two strings in the next two columns.

```
void GridAdapter::Append(wxString wname) {
    grid->AppendRows(1);           //append one row
    grid->DeselectRow(0);          //deselect
    that row
    //here is where to move it
    int rownum = grid->GetNumberRows() - 1;
    string name = wname.ToStdString();
    //separate at space
    vector<string>
        names = Strfuncs::split(name, " ");
    grid->SetCellValue(rownum, 0,
        wxString(names[0])); //first
    grid->SetCellValue(rownum, 1,
        wxString(names[1])); //last
    //get lengths of first and last name
    //last 2 cells are numbers
    grid->SetCellValue(rownum, 2,
        wxString(std::to_string(names[0].size()))
    );
}
```

```

grid->SetCellValue(rownum, 3,
wxString(std::to_string(names[1].size())));
}

```

Class Adapters

In this first, simple example, we simply moved string to the right and left. But in a more real-world application, we would probably have a class representing each student, and each instance would contain somewhat more useful student data.

Let's consider a simple Student class containing some scores and the first and last name:

```

class Student {
private:
    string name;
    string fname{ "" };
    string lname{ "" };
    int iq;
    int score;
public:
    Student(wxString nm);
    string getName();
    int getIQ();
    int getScore();
};

```

It would be nice if we could just add these Student objects to the ListBox and have the names displayed automatically. But as far as we know the only list box that could do that was in Visual Basic. So, it is up to us to create a class derived from wxListBox that appears to work that way. In fact, what we do is create a vector to hold the Student objects and clear and reload the listbox whenever the Student list changes.

The constructor just passes on the size and panel to the **wxList** constructor:


```
StudentList::StudentList(wxPanel* panel, wxSize size):
    wxListBox(panel, wxID_ANY, wxDefaultPosition, size,
              0, NULL, 0L, wxDefaultValidator) {}
```

Every time you add a Student to the list, it actually adds it to the vector and updates the listbox:

```
//append a new Student to vector
//and add a string to the list box
void StudentList::append(Student* st) {
    students.push_back(st);    //vector of students
    Append(st->getName());    //display name
}
```

If you delete a student from a list, the vector is updated and the listbox cleared and redrawn:

```
//remove a student from the vector and the list
void StudentList::remove(int index) {
    students.erase(students.begin() + index);
    redraw();
}
```

```
//redraw the listbox after any change
void StudentList::redraw() {
    Clear();
    for (int i = 0; i < students.size(); i++) {
        Append(students[i]->getName());
    }
}
```

And, if you want to get the currently selected Student from the listbox, you get its index and return the student at that index.

```
//return the student currently selected
Student* StudentList::getSelected() {
    int index = GetSelection();    //selection
    Student* std = students[index]; //get Student
    return std;
}
```

So, what we have done is create a class Adapter, derived from the basic wxListBox. Our demo program is made up of two

StudentList objects, each with their own internal vector. The calling program works just the same as the original one shown in Figure 22-2, except that it accesses two Adapter classes.

The GridAdapter class

In making an Adapter class for the we find that we can put the names in separate columns and actual data in the next two columns. We'll call these IQ and score (such as on a standardized test). Since we don't have any actual values for these fictitious students, we'll create them using a random number generator, that generates integers in a specified range. That generator is contained in a static method within our Student class:

```
class Student {
    //static random number generator
    static int randint(int min, int max) {
        int value = rand(); // rand in 0 to 1.0
        int range = value % (max - min); //range
        return range + min; //add min
    }
private:
    string name;
    string fname{ "" };
    string lname{" "};
    int iq;
    int score;
public:
    Student(wxString nm);
    string getName(); //get full name
    int getIQ(); //get IQ
    int getScore(); //get test score
    string getFname(); //get first name
    string getLname(); //get last name
};
```

Then, for each student, we generate a random IQ and score in the constructor and separate the name into two parts:

```

Student::Student(wxString nm) {
    score = Student::randint(25, 35); //score
    iq = Student::randint(115, 145); //IQ
    name = nm;
    //assume there are two names sep by a space
    vector<string> names =
        Strfuncs::split(name, " ");
    frname = names[0];
    if(names.size() >1)
        lname = names[1];
}

```

When we add a student to the grid, we add them to the **students** vector and then append that student to the grid:

```

//This append adds a student to the vector and then
adds it to the grid
void GridAdapterClass::append(Student* st) {
    students.push_back(st); //vector of students
    Append(st);
}

```

```

//This Append loads a row of the grid
void GridAdapterClass::Append(Student* st) {
    AppendRows(1); //append one row
    DeselectRow(0); //deselect that row
    //here is where to move it
    int rownum = GetNumberRows() - 1;
    SetCellValue(rownum, 0,
        wxString(st->getFrname())); //first
    SetCellValue(rownum, 1,
        wxString(st->getLname())); //last

    //last 2 cells are numbers-- show scores
    SetCellValue(rownum, 2,
        wxString(std::to_string(st->getIQ())));
    SetCellValue(rownum, 3,
        wxString(std::to_string(st->getScore())));
}

```

Deleting a line from the grid amounts to making sure it is not highlighted, removing it from the grid and then from the vector as well.

```

//Delete current row
void GridAdapterClass::Delete() {
    if (curRow >= 0) {
        //unselect row before deletion
        DeselectRow(curRow);
        DeleteRows(curRow, 1);
        remove(curRow); //delete from vector
    }
}

//remove student from vector
void GridAdapterClass::remove(int index) {
    students.erase(students.begin() + index);
    redraw();
}

```

Finding the current row

Unlike a list box, there is no concept of a current row, since you can select any group of cells you want. If you want to pass the click event from the Mediator on into the GridAdapter class, you can save the row from the event object.

```

//saves the currently clicked row
void GridAdapterClass::onClick(wxGridEvent& event) {
    curRow = event.GetRow(); //save current row
}

```

Or you can find it by running through all the cells in the first column to find the first one selected and saving that in that same **curRow** variable:

```

//find the currently selected row
int GridAdapterClass::GetSelection() {
    int found = -1;    int row = 0;
    //loop through cells in first row
    while (found < 0 && (row < GetNumberRows())) {
        if (IsInSelection(row, 0)) {
            found = row; //save that row
        }
        row++; //otherwise keep looking
    }
}

```

```
        if (found >= 0) {  
            curRow = found;           //save current row  
        }  
    return curRow;  
}
```

Object adapters and class adapters

The adapter we use in the string examples can be termed an *object adapter* because the GridAdapter code contains the grid and emulates a listbox. In the second case, we derive new classes from the wxListBox and the wxGrid that have methods much like those of the listbox. This is a minor distinction, but it does allow us to keep vectors of Student objects inside the class.

Example Code on GitHub

1. **BasicAdapter** – passes strings between two listboxes
2. **GridAdapter** -- passes strings between list box and grid.
3. **StudentListAdapter** – passes objects between two listbox classes
4. **StudentListGridAdapter** – passes objects between a listbox class and a grid class.

23. The Bridge pattern

At first sight, the Bridge pattern looks much like the Adapter pattern, in that a class is used to convert one kind of interface to another. However, the intent of the Adapter pattern is to make one or more classes' interfaces look the same as that of a particular class. The Bridge pattern is designed to separate a class's interface from its implementation, so that you can vary or replace the implementation without changing the client code.

More specifically, the Bridge is not between different visual interfaces as we saw with the Adapter patterns, but between the data and the visual representation.

Suppose that we have a program that displays a list of products in a window. The simplest interface for that display is a simple Listbox. But, once a significant number of products have been sold, we may want to display the products in a table along with their sales figures.

Since we have just discussed the adapter pattern, you might think immediately of the class-based adapter, where we adapt the interface of the Listbox to our simpler needs in this display. In simple programs, this will work fine, but as we'll see below there are limits to that approach.

Let's further suppose that we need to produce two kinds of displays from our product data, a customer view that is just the list of products we've mentioned, and an executive view which also shows the number of units shipped. We'll display the product list in an ordinary Listbox and the executive view in a Grid display. These two displays are the implementations of the display classes. We illustrate such a bridge in Figure 23-1.

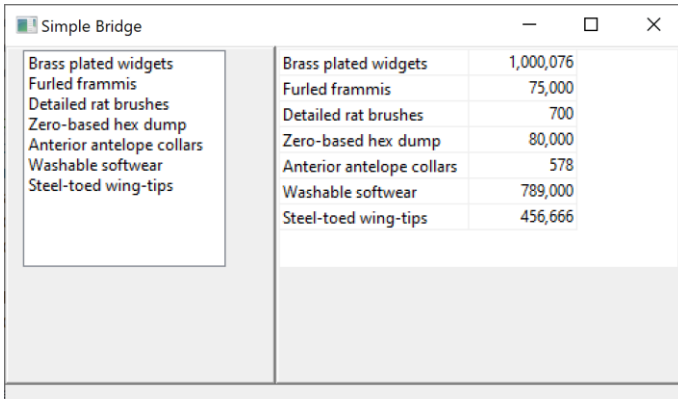


Figure 23-1 - Bridges between data and List and Grid displays

Our actual datafile just shows the text and quantities separated by two hyphens:

```
Brass plated widgets --1,000,076
Furled frammiss      --75,000
Detailed rat brushes --700
Zero-based hex dumps--80,000
Anterior antelope collars --578
Washable software   --789,000
Steel-toed wing-tips --456,666
```

We create two classes: Product which holds one entry and Products which holds a vector of Product instances.

```
class Product {
private:
    string name; //product name
    string count; //inventory or sales count
public:
    Product(const string nstring);
    string getName(); //return the name
    string getCount();//return the count
};
```

The actual code for Product parses each string into the product name and count:

```

Product::Product(const string nstring) {
    size_t index = nstring.find("--");
    name = nstring.substr(0, index - 1);
    count = nstring.substr(index + 2,
        nstring.length());
}
string Product::getName() {
    return name;
}
string Product::getCount() {
    return count;
}

```

The Products class reads each line, creates a Product instance and stores it in a vector:

```

class Products {
private:
    vector<Product*> prods; // list of products
public:
    Products(const string filename); //read file
    vector <Product*> getProducts(); //return vector
};

```

These two classes are the entire data side of the Bridge.

The Bridge

Now, we want to define a single, simple interface that remains the same regardless of the type and complexity of the actual implementation classes. We'll start by defining an abstract Bridger class:

```

//abstract Bridge class
class Bridger {
    //add data to the other side of the bridge
    virtual void addData(Products* prod) = 0;
};

```

This class is so simple that it just receives a List of data and passes it on to the display classes.

On the other side of the bridge are the implementation classes, which usually have a more elaborate and somewhat lower level

interface. Here we'll have them add the data lines to the display one at a time.

```
//abstract class defining the one method in the
VisList class
class VisList {
public:
    virtual void addLines(Products* prod)=0;
};
```

The Bridge between the interface on the left and the implementation on the right is the ListBridge class which instantiates one or the other of the list display classes. Note that it extends the *Bridger* class for use of the application program.

```
//General bridge between data and any VisList class
class ListBridge :public Bridger {
private:
    VisList* visList;
public:
    ListBridge(VisList* vl);
    void addData(Products* prod);
};
```

In the current example, we use the Bridge class twice: once to display the Listbox on the left side and once to display the Grid table on the right side.

The power and the simplicity of the Bridge pattern becomes obvious when you realize that you can completely change the display by replacing either or both of the two VisList classes that display the data. You don't have to change the Bridge class code: just give it new VisLists to display. And those classes can be anything, as long as they implement the simple VisList methods.

The VisLists

The VisList for Listbox is quite simple: it is derived from VisList and wxList.

```

//a listbox that holds a list of products,
//derived from VisList and ListBox
class LbVisList:public VisList,public wxListBox
{
public:
    LbVisList(wxPanel* p, wxSize sz);
    void addLines(Products* prods);
};

```

And the implementation of that class just adds lines to the ListBox:

```

//a VisList for a ListBox
//derived from wxListBox
LbVisList::LbVisList(wxPanel* p, wxSize sz):
    wxListBox(p, wxID_ANY, wxDefaultPosition, sz, 0,
        NULL, 0L, wxDefaultValidator) {
}
//add lines from the Products vector
void LbVisList::addLines(Products* prods) {
    vector<Product*> products = prods->getProducts();
    for (Product* p : products) {
        Append(wxString(p->getName()));
    }
}
}

```

The VisList for the Grid display is equally simple. Other than the Grid initialization code, it is almost exactly the same as for the ListBox:

```

//This is a Grid display which also has the VisList
interface "addLines"
GrdVisList::GrdVisList(wxPanel* p, wxSize sz) :
    wxGrid( p, wxID_ANY, wxDefaultPosition, sz) {
    CreateGrid(0, 2);
    SetColSize(0, 140); //set column sizes
    SetColSize(1, 80);

    SetRowLabelSize(0); //hide row label
    SetColLabelSize(0); //hide column labels
}

```

```

void GrdVisList::addLines(Products* prods) {
    vector<Product*> products = prods->getProducts();
    for (Product* p : products) {
        AppendRows(1);           //add one row

        //here is where to move it
        int rownum = GetNumberRows() - 1;
        SetCellAlignment(rownum, 1,
            wxALIGN_RIGHT, wxALIGN_CENTER);

        //insert the two values in columns 0 and 1
        SetCellValue(rownum, 0,
            wxString(p->getName())); //first
        SetCellValue(rownum, 1,
            wxString(p->getCount())); //last
    }
}

```

How to set up the Bridge

Once you see how simple it is to create these bridges in the main code, you will understand why Bridges are so useful. Here is all the code:

```

//create the listbox
leftList = new LbVisList(leftPanel, wxSize(150, 160));
leftSzc->Add(leftList);

//create right hand grid
rightList = new GrdVisList(rightPanel,
    wxSize(350, 160));
rightSzc->Add(rightList);

//create the products class and read in the data
prod = new Products("products.txt");

//create left bridge
lbridge = new ListBridge(leftList);
lbridge->addData(prod);

//create right bridge
rbridge = new GridBridge(rightList);
rbridge->add Data(prod);

```

Other VisLists

You can easily modify this design by creating a sorted ListBox and a sorted Grid. If they are all sorted in the same way, you could do the sorting in the Products class. But if each display is sorted differently, you would do the sorting in the VisList classes. Suppose we sort on VisList by name and the other by quantity as shown in Figure 23-2

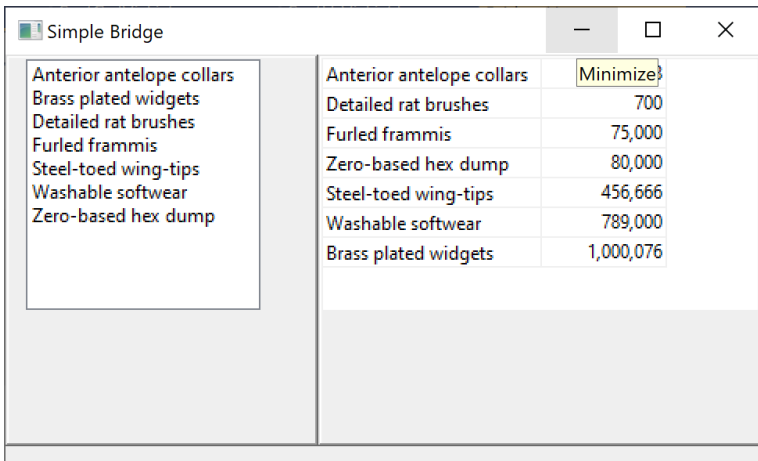


Figure 23-2 - Sorted Vislists using the same Bridge.

Then you would sort the ListBox by name by copying the array and sorting it in the SortLbVisList class:

```
void LbVisList::addLines(Products* prods) {
    vector<Product*> products =
        prods->getProducts();
    //sort the products by name
    vector <Product*> sortProd;
    //sort list by name
    //copy list
    for (Product* p: products) {
        sortProd.push_back(p);
    }
}
```

```

//sort list by name
for (int i = 0; i < sortProd.size(); i++) {
    for (int j = i; j < sortProd.size(); j++) {
        if (sortProd[i]->getName() >
            sortProd[j]->getName()) {
            swap(sortProd[i], sortProd[j]);
        }
    }
}
//now load the list with the sorted results
for (Product* p : sortProd) {
    Append(wxString(p->getName()));
}
}

```

Likewise, you can sort the list by count. However, our input data is all comma separated at the thousands and millions, and there is no simple parser to read these numbers. However, we can simply step through each string and skip any characters that aren't digits. This algorithm was suggested on the Runestone Academy site[22].

```

//convert comma space number into integer
int GrdVisList::getInteger(string s) {
    string digits = "";

    for (int i = 0; i < s.length(); i++) {
        char x = s[i];
        if (isdigit(s[i])){
            digits += s[i];
        }
    }
    return atoi(digits.c_str());
}
}

```

The sorting code is then;

```

//sort product list by count
for (int i = 0; i < sortProd.size(); i++) {
    for (int j = i; j < sortProd.size(); j++) {
        if (getInteger(sortProd[i]->getCount()) >
            getInteger(sortProd[j]->getCount())) {
            swap(sortProd[i], sortProd[j]);
        }
    }
}
}

```

Summary

The Bridge pattern provides a way to separate the data from the display cleanly so that you can modify either one without changing the other. Since this is a book about moving from Python to C++, this particular pattern can be written much more succinctly in Python because you need not declare variables or types. The advantage of C++ here is that this strong typing keeps you from making bizarre mistakes in managing coding between all of these abstract and concrete classes. So, while creating all this boiler plate code may seem tedious, overall, it makes your programming faster and more efficient.

Example programs on GitHub

1. Bridger1 — The bridges between the data and the ListBox and Grid.
2. SortBridges -- Bridges between sorted list and grid displays

References

1. https://runestone.academy/ns/books/published/thinkcpp/Chapter15/Parsing_numbers.html

Index

- ! logical Not, 61
- #include directives, 103
- #include statement, 29
- & reference operator, 75
- && logical And, 61
- * dereferencing pointer, 75
- || logical Or, 61
- == logical equals, 61
- Abstract Bridger class, 304
- Abstract class
 - or interface, 270
- Abstract Factory, 284
- Abstract functions, 109
- accelerator character in
 - Menus, 156
- Adapter pattern, 290
- Adding rows to a Table, 236
- Adding two numbers in
 - wxWidgets, 149
- AI systems, 18
- Alignment
 - in wxWidgets, 143
- apostrophe
 - as digit separator, 115
- Arguments
 - in functions, 72
- Arithmetic operations, 27
- Arithmetic shortcuts, 28
- Armadillo math library, 187
 - installation, 197
 - overview, 187
- Array of char, 46
- Arrays
 - and pointer, 76
 - declaring contents, 49
 - printing, 50
 - two dimensional, 50
- auto keyword, 32
- Binary file
 - reading, 47
- Binary files, 46
- Bind, 176
 - wxWidgets events, 148
- Binding MenuItems, 157
- Bitwise operators, 30
- BlueLabel class
 - in wxWidgets, 145
- Book organization, 21
- bool type, 25
- Box sizer, 142
- brace construction, 94
- brace notation
 - to initialize class
 - variables, 90
- braced initializers, 94
- braces
 - in conditions, 60
 - in functions, 29
 - positions of, 31
- Braces
 - in classes, 90
- break**
 - and **continue**, 65
- break** statement, 63
- Bridge Design Pattern, 302
- Bridge pattern, 302
- Builder class

- builds wxWidgets windows, 143
- C strings, 55, 80
- C++ array type, 49
- C++ development systems, 33
- Call by reference, 77
- Call by value, 77
- Cell class, 122
- char**
 - array, 46
- char type, 25
- character constants, 26
- Character Constants, 26
- Checkbox styles, 177
- CheckBoxes, 175
- CheckListBoxes, 170
- Choices and Listboxes, 163
- cin
 - getline** method, 38
 - cin object, 38
 - converts to correct type, 38
- class**
 - in templates, 131
- class adapters, 290
- Class Adapters, 296
- Classes, 89
 - braces, 90
 - constructor, 90
 - deriving, 96
 - end with semicolons, 90
 - inheritance, 92
 - methods, 89
- Classes and headers, 100
- CLion, 35
- CodeBlocks, 34
- Collections
 - in MySQL, 257
- Colors
 - in **wxWidgets**, 144
- combing conditions, 61
- combining conditions, 61
- ComboBox, 174
- Command Button, 152
- Command Design Pattern, 154
- Comments
 - multi-line, 24
 - single line, 24
- comparing strings, 62
- conditions
 - combining, 61
- Connector C++
 - Table functions, 256
- Connector C++ code, 244
- const** declarations
 - in functions, 74
- Constant classes, 112
- constants
 - character, 26
- constructor
 - in classes, 90
- Constructor, 90
 - default value, 94
- continue**, 65
- Copy constructor, 126
 - deleting, 128
 - syntax, 127
- cout object, 29, 37
- Curve fitting
 - in Armadillo, 195
- Data encapsulation, 89
- data** method
 - creating mutable C-string, 80
- Data types, 25, 190
- Database

- Column class, 234
- Database classes, 229
- Database tables, 233
- Databases, 219
- dbMap, 252
- DButton abstract class, 152
- Default arguments
 - in functions, 73
- Default value
 - in constructor, 94
- delete** method, 116
- Deriving new classes, 96
- Design patterns, 269
- Design Patterns
 - Abstract Factory pattern, 284
 - Adapter Pattern, 290
 - Bridge Pattern, 302
 - Command, 154
 - Facade, 249
 - Factory Method Pattern, 276
 - Mediator, 172
 - Simple Factory Pattern, 273
- destructor method
 - class, 117
- Dialog Boxes, 158
- do while** loop, 52
- double
 - number to string, 58
- double type, 25
- doubly linked list, 121
- elif
 - else if, 60
- else** clause, 60
- else if** clause, 60
- Employee class, 93
- Encapsulation, 270
- endl object, 29, 37
- enum**, 155
- Error bars
 - in ROOT, 213
- Events in wxWidgets, 147
- Facade pattern
 - for databases, 249
- Factory Method Pattern, 276
- Factory Patterns, 273
- File Dialog, 159
- File handling, 45
- Files
 - binary, 46
- float type, 25
- for** loop, 49, 51
 - range based, 51
- Format function
 - error handling, 44
 - symbols in, 42
- Formatting
 - in C++, 40
 - in Python, 39
- Fourier transform, 193
- Frame
 - in wxWidgets, 139
- friend declarations, 112
- Function
 - indentation, 69
- Function prototypes, 71
- functions, 69
 - polymorphism, 71
- Functions
 - abstract, 109
 - arguments, 72
 - call by reference, 77
 - call by value, 77
 - default arguments, 73
 - order, 70

- Gang of Four
 - GoF book, 269
- GardenMaker Factory, 284
- GitHub, 18
- Grid Adapter, 294
- Grid display widget, 281
- GridAdapter class, 298
- GridBag sizer, 149
- head and tail pointers, 122
- Headers
 - for classes, 100
- heap, 115
- Hexadecimal, 43
- hidden copy constructor, 126
- if** statement, 60
- ifstream object, 45
- Include files
 - in wxWidgets, 142
- include statement, 29
- Indentation
 - in functions, 69
- Inheritance, 92, 270
 - multiple, 105
 - private, 97
 - protected, 97
 - public, 97
 - public or private, 97
- int type, 25
- integer
 - number to string, 58
- integers
 - lengths, 31
- Interface, 270
- iosteam library, 29
- isChecked** method, 157
- iterator
 - in linked list, 124
- Labels, 144
- wxStaticText**, 140
- LAPACK, 187
- Linked list
 - inserting, 125
 - iterator, 124
- Linked lists, 121
- LinkedList class, 123
- ListBoxes, 168
 - Append method, 169
 - Check, 170
 - finding the selection, 169
 - loading from string list, 169
 - multi-select, 169
 - style settings, 168
- ListBridge class, 305
 - Bridge pattern, 305
- long long type, 31
- long type, 31
- main function, 29
- making decisions, 60
- map** object, 258
- map** type, 87
- Maps, 87
 - fetching by key, 87
- Math constant pi, 133
- MatPlotLib, 204
- Matplotlibplusplus, 204
- Matrices, 188
 - columns and rows, 190
 - creating, 188
- Matrix methods
 - in Armadillo, 190
- Matrix transpose, 191
- Mediator, 288
- Mediator class, 172
- memory leaks, 118
- MenuItems, 155
- Menus in wxWidgets, 154

- Merging sets, 84
- Method section
 - when using prototypes, 102
- Methods
 - in classes, 89
- Modules, 263
 - in a Visual Studio project, 264
- modulo, 27
- most common mistake
 - = instad of ==, 61
- Multiple inheritance, 105
- Multi-select ListBoxes, 169
- MySQL, 243
 - installation, 243
 - numeric types, 251
- MySQL C++ classes, 248
- MySQL programming
 - in Visual Studio, 245
- MySQL query
 - using Connector C++, 246
- MySQL Workbench, 244
- NameFactory, 275
- Namespaces, 261
- newline, 26
- object adapters, 290
- Object adapters, 301
- object composition, 270
- Object composition, 270
- ofstream object, 45
- onClick event method, 148
- OnlineGDB
 - development system, 33
- OpenBLAS, 187
- Order
 - of functions, 70
- Other VisLists, 308
- Plotting in C++, 201
 - using DrawLinesin wxwidgets, 203
 - using wxWidgets, 201
- pointer
 - dereferencing, 75
- Pointer, 75
- Pointers
 - as function arguments, 77
 - in vectors, 98
- polymorphism
 - functions, 71
- Polymorphism, 107
- pop_back**
 - in vectors, 53
- port 33060
 - instead of 3306, 246
- pragma once, 104
- Prepared Queries, 238
- Prepared Queries in MySQL, 254
- Private inheritance, 97
- private** variables, 90
- Program to an interface, 270
- Protected inheritance, 97
- protected** variables, 90
- Prototypes
 - and functions, 71
 - in header, 102
- Public inheritance, 97
- public** methods and variables, 93
- public** section
 - of class, 93
- Pure virtual functions, 109
- push_back**
 - in vectors, 52
- Python
 - types in, 25

- RadioButtons, 163
 - finding the calling object, 167
 - finding which is selected, 164
 - responding to clicks, 165
- Rectangle class, 89
- reference operator
 - ampersand, 75
- Relational databases, 219
- return** statement, 69
- Reverse iterator
 - in linked list, 125
- ROOT interpreter, 208
- ROOT plotting package, 207
 - error bars, 213
 - for a C compiler, 212
 - marker styles, 210
 - writing C++ code, 211
- SciPlot, 205
 - Vec, 205
- Selecting Grid regions, 180
- semicolon, 23
 - at end of class, 90
- semicolons
 - at the end of classes, 90
- Sets, 83
 - find method, 83
 - merging, 84
 - of strings, 84
 - tie** function, 86
- short type, 31
- Shortcuts
 - arithmetic, 28
- Simple C++ program, 29
- Simple Factory Pattern, 273
- singly linked list, 121
- size_t type, 27
- Sizers
 - Layout managers, 142
- Smart pointers, 118
- SQL, 220
- SQLite
 - compiling using Visual Studio, 223
 - installing, 221
 - programming in C++, 223
 - Results class, 232
 - SQLite database, 221
 - SQLite queries
 - callback structure, 230
 - Sqlite3 prepared statement interface
 - eliminates callback, 239
 - sqlite3_open**, 230
 - sqlite3_open_v2**, 230
 - sqlitePrepQuery
 - eliminates callback, 241
- Square class
 - derived from Rectangle, 92
- Standard Template Library, 129
- statements, 23
- static class members, 111
- std library, 29
- std namespace, 39
- stod
 - string to double, 58
- stof
 - string to float, 58
- string
 - insert**, 56
 - range based access, 58
- string type, 25
- strings, 55

- begin, 57
- combining, 55
- comparing, 62
- end, 57
- replacing characters, 56
- reversing in place, 56
- to numbers, 58
- strongly typed, 24
- Structured Query Language, 220
- swap** function, 129
- Swimmer class, 277
- switch** statement, 63
- Template functions, 130
- Templates, 129
- Templates of classes, 132
- ternary operator, 67
- The Adapter pattern, 290
- this** pointer, 95
 - in classes, 95
- tie** function, 86
 - in sets, 86
- to_string
 - number to string, 58
- ToStdString**
 - in **wxString**, 140
- Tree widget, 182
- Tuples, 85
 - retrieving values by index, 85
 - tie** function, 86
- typed
 - strongly, 24
- typename**
 - in templates, 131
- unique_ptr**'s, 118
- unordered_set**, 83
- unsigned type, 31
- using** declaration, 38
 - using namespace, 29
- using** statements, 39
- Value object
 - in query return, 253
- Variable names, 23
- Variables
 - declaring, 24
- Vec
 - in SciPlot, 205
- vector
 - methods, 54
- Vector
 - of pointers, 98
- vectors, 52
 - creating a new, 54
- Vectors
 - passing to functions, 79
- Virtual functions, 108
 - pure, 109
- VisList abstract class, 305
- Visual Studio, 34
- while** loop, 52
- wxButton, 146
- wxGBSpan method, 150
- wxGrid** widget, 178
- wxListBox, 292
- wxMenu, 154
- wxMenuBar, 154
- wxPanel**, 140
- wxPoint**, 141
- wxPython
 - related to wxWidgets, 138
- wxRadioButtons**, 163
- wxString**, 140
- wxT**, 140
- wxWidgets, 137, 292
 - colors, 144
 - ComboBox, 174, 175

events, 147
Frame, 139
include files, 142

wxPanel, 140
wxTreeCtrl, 182

