C++ Course Guide

Composed by: Kerrie Lee Harris and Willy Susilo
School of Information Technology & Computer Science
University of Wollongong, Australia

This document is to accompany UoW programs only: Bachelor of IT (Computing), Bachelor of Computer Science (Digital System Security specialisation) and Bachelor of Computer Science (Multimedia and Game Development specialisation).

Preface

This manual is designed as an introduction to the C++ programming language. It offers a basic understanding of C++ programming concepts and syntax which, enables the student to further study the content at a higher level.

In order to complete this manual the student will require access to a compiler and a simple text editor. Word processing packages are not suitable. A link to the Quincy Integrated Development Environment which contains both the compiler and editor can be found below.

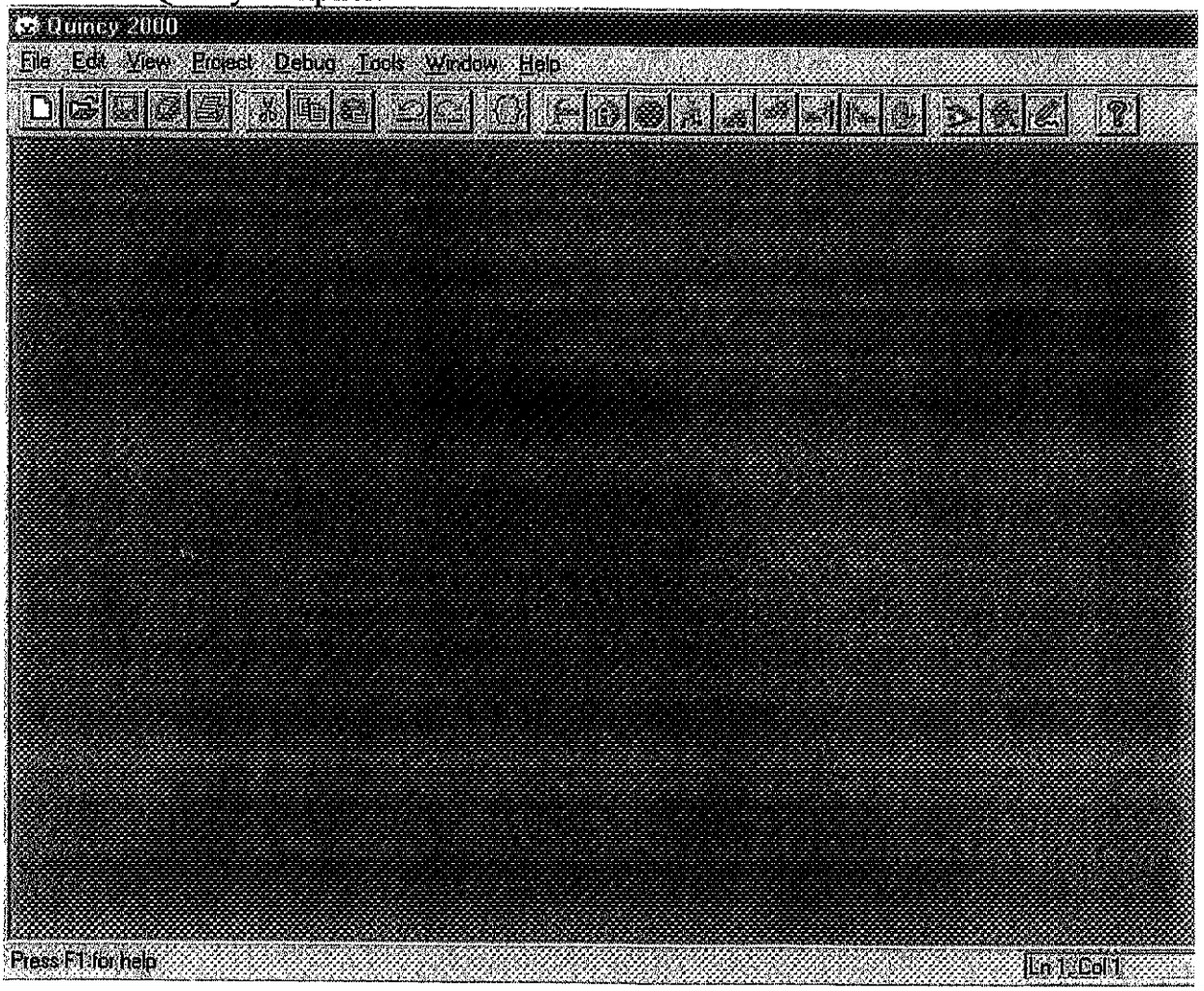
Use the following link to download Quincy.
http://www.sitacs.uow.edu.au/subjects/csci114/Quincy2005setup.exe
UserName: csci114
Password: willy

Quincy - Step by Step

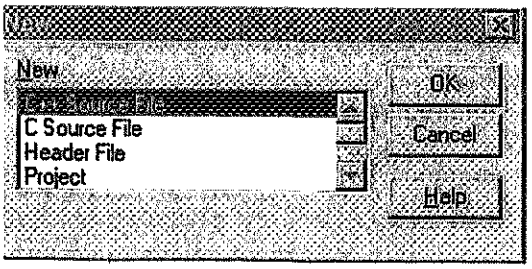
How to create & compile a .CPP program with Quincy?

This simple tutorial will show you step-by-step on how you can create and compile a .CPP program with Quincy.
You need to install Quincy in your computer and follow the following tutorial.
Let's assume that you will write a very simple *"Hello World"* program.

1. Run the Quincy compiler.

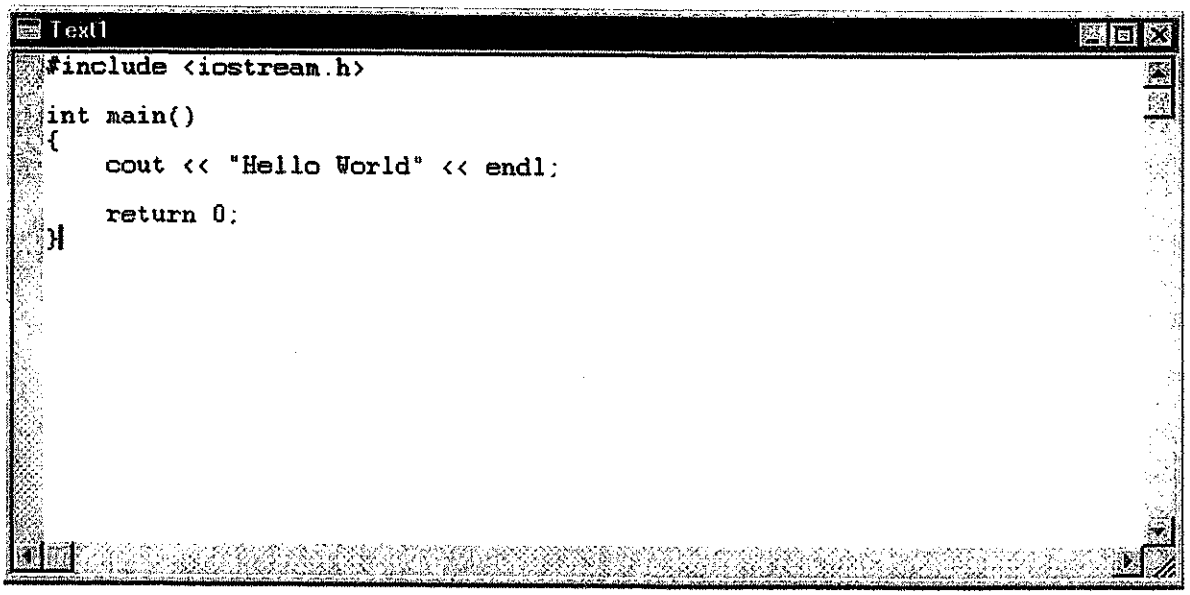


2. Click "File-New" to create a new file.

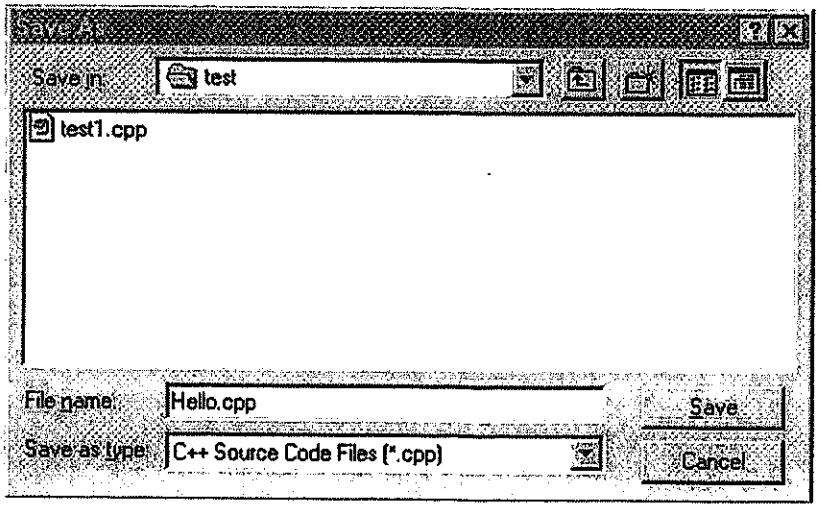


3. Select "C++ Source File" and click "OK".

4. Type the "C++ code" in the window. For example,

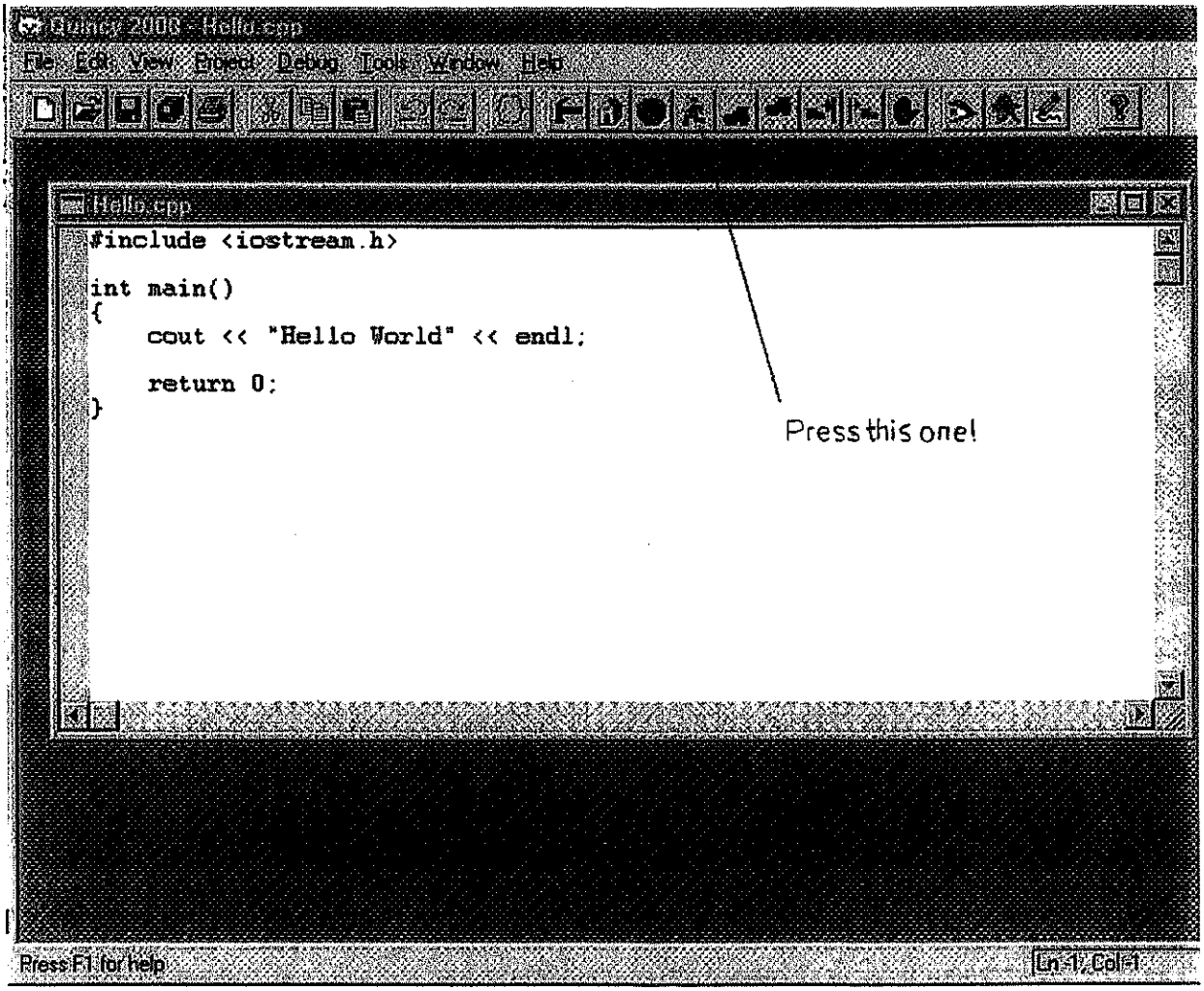


5. Select "File-Save As", and write the file name. In this example, I use "Hello.cpp". Note that the file name MUST be ended with ".CPP".

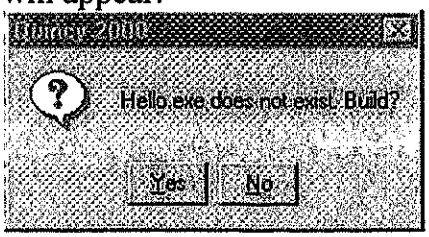


6. Select "Save".

7. Try to compile the program by selecting the icon which "a man running".

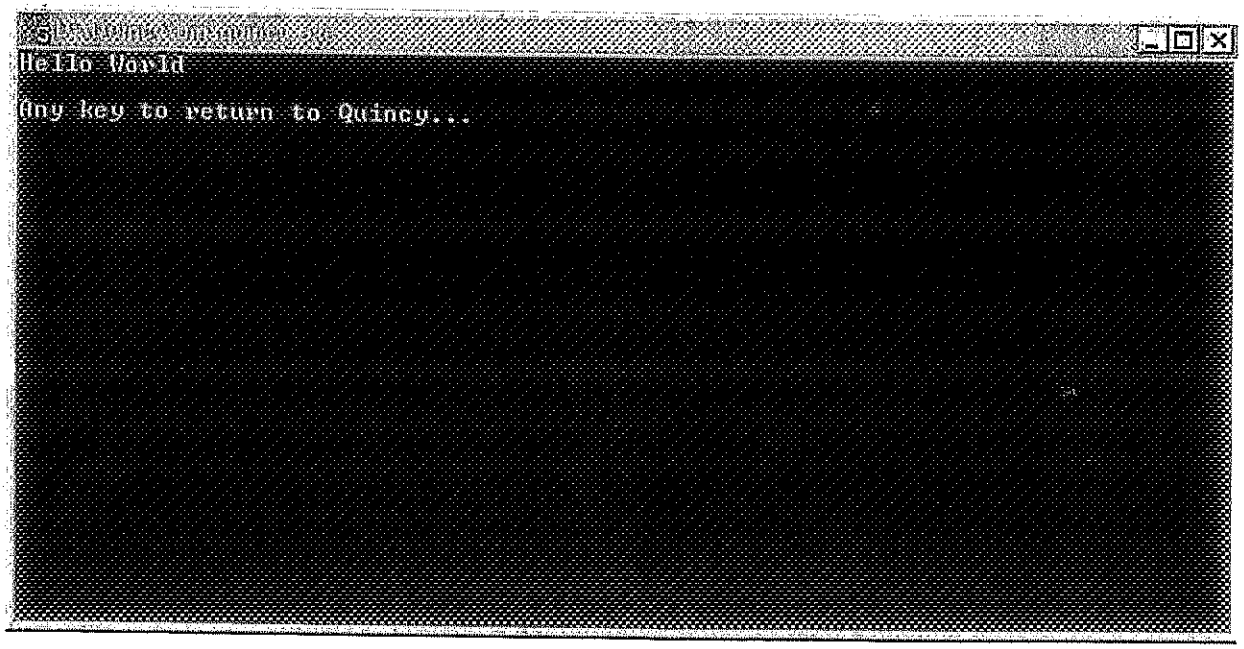


8. Next, the following dialog will appear:



Press "Yes" to confirm to build the executable code.

9. Finally, you will see the output of your code.


Hello World
Any key to return to Quincy...

Basic Elements of C++

What is a Program?

A program is an organized list of instructions that when executed causes the computer to behave in a predetermined manner.

The general purpose of a program is to:

- Receive input.
- Perform some calculation/manipulation of the input data.
- Produce output.

Basics of a C++ Program

Every C++ program contains one or more functions, which will be explained in detail later. One of those functions must be the `main` function which is where the program beings execution.

Each function contains a list of instructions, called *statements*, to be executed when the program is run. A block of statements is referred to as *code*. The program in its entirety is called the source code. The code is written in a C++ source file, which typically has the file extension "`.cpp`"

Create a source file for a C++ program named "`basicProgram.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

The name of the primary function in a C++ program is `main` and it always returns an integer value to the operating system. The returned value indicates whether the program terminated successfully. If so, the value returned is 0 otherwise it is 1.

The code:

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

Will enable the use of input and output operations which will be discussed in detail later.

```
Compile and run the program.
```

This program does compile and run, although it does nothing at present. The only executable code in the program is:

```
return 0;
```

This is the last statement in any C++ program and is an instruction to return the value `0` to the operating system indicating successful completion of the program.

Data Types

A data type refers the *type* of the data being stored in memory. (I.e. An integer, a character or a floating point value) C++ has several built-in data types which are allocated a certain number of bytes to hold its data. The amount of memory allocated is dependent upon the system running the application, but the common values are:

- `int` – 4 bytes
  - Can hold an integer value between `2,147,483,647` and -2,147,483,648.
- float – 4 bytes
  - The range of the numbers and the precision is platform dependent. However, the largest and smallest numbers will be sufficient for this manual. Likewise, the precision.
- `double` – 8 bytes
  - As above.
- `char` – 1 byte
  - Actually, it is the ascii value of the character which is stored and this is an integer value between 0 and 127. Any ascii value can be stored, regardless of whether it is an alphabetic character, a numeral, or a non-printable character. The uppercase alphabetic characters are in the range 65 - 90 (A – Z) while the lowercase characters are in the range 97-122 (a-z).
- `bool` – 1 byte
  - Used to store the Boolean values 1(true) and 0(false). These are the only two values that can be assigned to a `bool` type. Two constants, `true` and `false`, have been defined to be used with `bool` types.

Variable Declaration Statement

Programs most often require data to be stored in memory for manipulation. To allocate memory, a declaration statement is required to create a *variable*. That is, assign memory for a data item whose value can vary (change). The variable declaration statement requires the `type` of the variable and the name to which the memory location will be linked.

Add the following code to `main`:

```
int iVar;
float fVar;
```

```
char chVar;
```

To:

- Declare three variables of type `int`, `float` and `char`.

C++ variable names must conform to certain rules. They may only begin with the characters `[a-z]`, `[A-Z]` and `[_]` and only contain the characters `[0-9]`, `[a-z]`, `[A-Z]` and `[_]`.

Upon executing a variable declaration statement, the compiler allocates the required amount of memory and links the name of the variable (*which the programmer uses*) to the address of the memory (*which the compiler uses*). The programmer only needs to know the name of the variable in order to access its associated memory.

To declare more than one variable in a single statement, all must be of the *same* `type` and the variable names need to be separated by the comma (`,`) operator.

Modify the following code:

```
int iVar;
```

To be:

```
int iVar, iVar2;
```

To:

- Declare two variables of type `int`.

```
Compile the program.
==========================================================
[Warning] unused variable 'iVar'
[Warning] unused variable 'iVar2'
[Warning] unused variable 'fVar'
[Warning] unused variable 'chVar'
==========================================================
Note:
Warnings should be produced. The compiler option (-Wall) to
display all warnings must be set to see ALL warnings,
otherwise some warnings may not be displayed.
```

The warnings produced are informing that the variables `iVar`, `iVar2`, `fVar` and `chVar` have not been used in the program.

Warnings are an indication that there may be a potential logic error in the code. In this case it can be ignored as the variables will be used shortly. In general though, code producing warnings should be modified to remove the warning(s).

Assignment Statement

Now that variables have been declared and the corresponding memory allocated, values can be assigned (*given, set*) to those memory locations. This is achieved by use of the assignment (=) operator.

Add the following code to `main`:

```
iVar = 100;
fVar = 100.5;
chVar = 'A';
```

To:

- Assign values to the three variables.

Note that the statement:

```
iVar = 100.0;
```

Is *not* the same as:

```
100.0 = iVar;
```

The latter code would not compile as values can *only* be assigned to identifiers.

The single quote (') is required on either side of **A** to distinguish a character from a variable name.

I.e. **'A'** is a character, while **A** is a variable name

```
Compile the program.
=============================================================
Note:
The warning regarding the unused variable iVar2 will still be
produced and can still be ignored. Ideally, if the variable is
not required its declaration should be removed.
```

The program should compile successfully.

Add the following code to `main`:

```
cout << "iVar  = " << iVar << endl;
cout << "fVar  = " << fVar << endl;
cout << "chVar = " << chVar << endl;
```

To:

- Output the values of the three variables.

Note: This code produces output which will be discussed in detail later.

```
Compile and run the program.
```

```
iVar  = 100
fVar  = 100.5
chVar = A
```

```
Note:
The warning about an unused variable will still be produced
and can still be ignored (or the variable declaration for
iVar2 can be removed).
```

Variables can also be assigned the values of other variables.

Add the following code to `main`:

```
iVar2 = iVar;
cout << "iVar2 = " << iVar2 << endl;
iVar = iVar + 1;
cout << "iVar  = " << iVar << endl;
```

To:

- Assign the value of `iVar` to `iVar2` and output `iVar2`'s value, then increment `iVar1` and output its value.

In the statement:

```
iVar = iVar + 1;
```

The value in `iVar` is retrieved from memory and `1` is added to this value, then the result is assigned to `iVar`.

```
Compile and run the program.
```

```
iVar  = 100
fVar  = 100.5
chVar = A
iVar2 = 100
iVar  = 101
```

```
Close the file.
```

Expressions

An expression is a combination of values, variables, operators, and grouping symbols (parentheses `()`) that are interpreted (*evaluated*) according to the particular rules of precedence and of association for a particular programming language. That is, the order in which the expression is evaluated is also dependent upon the language used. The result of the expression, the evaluation, is another value.

Create a source file for a C++ program named "`integerExpressions.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

The simplest expression is a single value.

Add the following code to main:

```
cout << "Integer operands: " << endl;
cout << "    0 = " << 0 << endl;
```

To:

- Output to screen the value of `0`.

Expressions using Arithmetic Operators

The standard arithmetic operators can be used to manipulate integral and floating-point data types. There are five arithmetic operators:

1. \+ Addition
2. \- Subtraction
3. * Multiplication
4. / Division
5. % Remainder (modulus Operator – applicable to integral types only)

All five are binary operators; that is, they require two operands. However, two of the operators are **also** unary operators:

1. \+ positive
2. \- negative

Whenever + or – are used with only one operand, on the right, they will be interpreted as the unary operator.

The rules applying to the order of precedence to the arithmetic operators are the same as in any mathematical expression.

Arithmetic Expressions with Integer Operands

When the C++ arithmetic operators are given integer operands, the result of the expression will also be of an integer type.

Add the following code to `main` which uses the unary arithmetic operators:

```
cout << "   -6 = " << -6 << endl;
cout << "   +6 = " << +6 << endl;
```

To:

- Output to screen the values `-6` and `6`.

Add the following code to main, which uses the binary arithmetic operators:

```
cout << "6 + 4 = " << 6 + 4 << endl;
cout << "6 - 4 = " << 6 - 4 << endl;
cout << "6 * 4 = " << 6 * 4 << endl;
cout << "6 / 4 = " << 6 / 4 << endl;
cout << "6 % 4 = " << 6 % 4 << endl;
```

To:

- Output to screen the values of several expressions using the binary arithmetic operators.

```
Compile and run the program.
```

```
    0 = 0
   -6 = -6
   +6 = 6
6 + 4 = 10
6 - 4 = 2
6 * 4 = 24
6 / 4 = 1
6 % 4 = 2
```

Notice the result of the last two expressions.

```
6 / 4 = 1
6 % 4 = 2
```

When the divide (`/`) operator is used with integer operands, integer division is performed. Integer division is division in which the fractional part (remainder) is discarded. On the other hand, when the modulus (`%`) operator is used with integer operands, the remainder of a divide operation is the result. Hence, `4` divides `6 1` time with a remainder of `2`.

```
Close the file.
```

Arithmetic Expressions with Floating-Point Operands

When the C++ operators are given floating-point operands, the result of the expression will also be of a floating-point type.

Create a source file for a C++ program named "`floatingPointExpressions.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Add the following code to `main`:

```
cout << setiosflags (ios::fixed | ios::showpoint)
     << setprecision (1);
cout << "Floating-point operands:" << endl;
```

To:

- Format the output for floating-point values.

Add the following code to `main`, which uses the unary arithmetic operators:

```
cout << "     -6.0 = " << -6.0 << endl;
cout << "     +6.0 = " << +6.0 << endl;
```

To:

- Output to screen the values of expressions using the unary arithmetic operators.

Add the following code to `main`, which uses the binary arithmetic operators:

```
cout << "      0.0 = " << 0.0 << endl;
cout << "6.0 + 2.0 = " << 6.0 + 2.0 << endl;
cout << "6.0 - 2.0 = " << 6.0 - 2.0 << endl;
cout << "6.0 * 2.0 = " << 6.0 * 2.0 << endl;
cout << "6.0 / 2.0 = " << 6.0 / 2.0 << endl;
cout << "6.0 % 2.0 = " << 6.0 % 2.0 << endl;
```

To:

- Output to screen the values of expressions using the binary arithmetic operators.

```
Compile the program.
```

```
floatingPointExpressions.cpp:: error: invalid operands of
types 'double' and 'double' to binary 'operator%'
```

```
Note:
The program will not compile.
```

The program does not compile because of the statement:

```
6.0 % 2.0
```

This statement is illegal in C++ as the modulus (%) operator is *only* applicable to integer types. Delete the following line of code to remove the error.

```
cout << "6.0 % 2.0 = " << 6.0 % 2.0 << endl;
```

```
Compile and run the program.
```

```
Integer operands:
    0 = 0
   -6 = -6
   +6 = 6
6 + 4 = 10
6 - 4 = 2
6 * 4 = 24
6 / 4 = 1
6 % 4 = 2
Floating-point operands:
     -6.0 = -6.0
     +6.0 = 6.0
      0.0 = 0.0
6.0 + 2.0 = 8.0
6.0 - 2.0 = 4.0
6.0 * 2.0 = 12.0
6.0 / 2.0 = 3.0
```

```
Close the file.
```

Arithmetic Expressions with Mixed Operands and Implicit Type Conversion

A requirement of C++ arithmetic expressions is that both operands for each arithmetic operator be the same type. If the operands are different the compiler will convert one side of the operation to the type of the other side. This is known as type *coercion.* The conversions possible for arithmetic expressions are:

- `char` → `int / float / double`
- `int` → `float / double`
- `float` → `double`

Create a source file for a C++ program named "`mixedExpressions.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

The include statement below allows the use of formatting functions for output operations.

```
#include <iomanip>
```

Add the following code to `main`:

```
cout << setiosflags (ios::fixed | ios::showpoint)
     << setprecision (1);
```

To:

- Allow formatted output (*to be discussed in detail later*).

Add the following code to `main`:

```
cout << "6.0 + 4   = " << 6.0 + 4
     << endl;
cout << "6.0 / 4   = " << 6.0 / 4
     << endl;
cout << "'a' + 1   = " << 'a' + 1
     << endl;
cout << "'a' + 1.5 = " << 'a' + 1.5
     << endl;
cout << "'a' / 1.5  = " << 'a' / 1.5
     << endl;
```

To:

- Output to screen the values of expressions using mixed operands on the binary arithmetic operators.

```
Compile and run the program.
```

```
6.0 + 4   = 10.0
6.0 / 4   = 1.5
'a' + 1   = 98
'a' + 1.5 = 98.5
```

```
'a' / 1.5 = 64.7
```

The result of the third expression is 98, which is the ascii value of 'a' (97) + 1. Likewise, the expression ['a' + 1.5] returns 98.5 as the ascii value of 'a' (97) was converted to a double (97.0) and added to 1.5.

```
Warning: While the compiler will perform the conversions
unquestioningly, it is up to the programmer to ensure that the
expression is logical.
```

```
Close the file.
```

Assignment Statements and Implicit Type Conversion

Type coercion also occurs during an assignment statement. In this case the type of the value being assigned will be converted to the type of the variable.

Create a source file for a C++ program named "`assignmentCoercion.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

The conversions possible for assignment statements are:

- `char` ← `int`
- `int` ← `char / float / double`
- `float` ← `int / double / char (ascii value)`
- `double` ← `int / float / char (ascii value)`

Add the following code to `main`:

```
float fVar = 1;
```

To:

- Declare a variable `float` variable and assign an `int` value to it.

Likewise, the compiler can convert floating-point types to integer types similarly.

Add the following code to `main`:

```
int iVar = 1.5;
```

To:

- Declare an `int` variable and assign to it a floating-point value.

```
Compile the program.
=============================================================
[Warning] converting to `int' from `double'
```

The warning is produced because a `double` type is being assigned to an `int` type. The compiler will resolve this warning itself by truncating the value being assigned. That is, discarding the decimal component.

Add the following code to `main`:

```
cout << "fVar = " << fVar << endl;
cout << "iVar = " << iVar << endl;
```

To:

- Output the values of `fVar` and `iVar`.

```
Compile and run the program.
=============================================================
fVar = 1
iVar = 1
```

Notice that the value `1.5` has been truncated to `1` before being assigned to `iVar`. The `.5` has disappeared. Also, even though the output for `fVar` is `1` and not `1.0`, its value is certainly a floating-point value. Formatted output will be introduced shortly to allow the value of a `float` to be shown with the decimal point.

```
Close the file.
```

Programming Exercise:
Perform some simple arithmetic calculations and output results.

1. Create a C++ program named "`practiceExerciseBasics.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Define the main function and do the following:
4. Declare two `int` variables, giving them some initial value (this can be whatever you like).
5. Declare two `float` variables, giving them some initial value.

6. Add the appropriate output statements as seen above to output the value of the four variables.
7. Declare a third `int` and third `float` variable to hold the result of some simple arithmetic calculations performed on the respective types.
8. Perform the following calculations on the `int` variables and assign the value of the expression to the result variable then output its value in a `cout` statement as shown above.
   a. Add the two variables.
   b. Output the result.
   c. Subtract the two variables.
   d. Output the result.
   e. Multiply the two variables.
   f. Output the result.
   g. Divide the two variables.
   h. Output the result.
   i. Mod % the two variables.
   j. Output the result.
9. Perform the following calculations on the `float` variables and assign the value of the expression to the result variable then output its value in a `cout` statement as shown above.
   a. Add the two variables.
   b. Output the result.
   c. Subtract the two variables.
   d. Output the result.
   e. Multiply the two variables.
   f. Output the result.
   g. Divide the two variables.
   h. Output the result.
10. Compile and run the program to ensure that the code is logically correct. That is, it performs the required task.

Explicit Type Conversion

Type coercions are performed automatically by the compiler without intervention by the programmer and are called *implicit* conversions. The programmer may also convert a value from one type to another. This is called *casting* and is an *explicit* conversion.

Create a program named "staticCast.cpp" and add the following code:

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

int main ()
{
```

```
    return 0;
}
```

To:

- Define the `main` function.

The cast (`static_cast<>()`) operator is available to explicitly cast one type to another and can be used to cast any of the built-in types to another built-in type. The type to which the data is to be cast is placed *inside* the **<>** brackets and the value to be converted is placed *inside* the **()** brackets. It is important to note that the value of the parameter, inside the **()** brackets is *never* altered.

Add the following code to `main`:

```
float fVar = 10.5;
int iVar = static_cast<int>(fVar);
cout << "fVar = " << fVar << endl;
cout << "iVar = " << iVar << endl;
```

To:

- Declare a `float` variable and set its value and declare an `int` variable and assign to it the cast value of the `float` variable.

```
Compile and run the program.
============================================================
10.5
10
```

When using the `static_cast<>()` operator, warnings about type conversions are no longer displayed and the variable `fVar` itself is never altered. The value of the parameter is retrieved, converted to the required type and this is returned by the operation (and hence used accordingly).

A cast may also be performed on `char` data.

Add the following code to `main`:

```
char chVar = 'a';
cout << "chVar = '" << chVar << "' as an int = "
     << static_cast<int>(chVar) << endl;
```

To:

- Declare a `char` variable and assign it the value `'a'` then output to screen the cast value of it as an int.

It is important to understand that the `char` type is stored internally as an integral value. That is, the *ascii* value of the character is stored in 8bit binary. Thus, in the statement:

```
char chVar = 'a';
```

An implicit type conversion occurs and the `'a'` is converted to an `int` and the value `97` is assigned to `chVar`.

Add the following code to main:

```
chVar = 98;
cout << "chVar = '" << chVar << "' as an int = "
     << static_cast<int>(chVar) << endl;
```

To:

- Assign the value `98` to `chVar` and output to screen the cast value of it as an int.

```
Compile and run the program.
=============================================================
fVar = 10.5
iVar = 10
chVar = 'a' as an int = 97
chVar = 'b' as an int = 98
```

Although `char` variables store data using the *ascii* value (integer), when outputting the value, actual characters are produced.

Programming Exercise:
Write a program to perform some simple arithmetic calculations with mixed operands and output the results.

1. Create a C++ program named "`practiceExerciseBasicsToo.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Define the main function and do the following:
4. Declare one `int` variable, giving it some initial value (this can be whatever you like).
5. Declare one `float` variable, giving it some initial value.
6. Add the appropriate output statements as seen above to output the value of the two variables.
7. Declare a third `float` variable to hold the result of some simple arithmetic calculations performed on the variables.
8. Perform the following calculations using the `int` variable as one operand and the `float` variable as the other and assign the value of the expression to the result variable then output its value in a `cout` statement as shown above.
9. a. Add the two variables.
   b. Output the result.
   c. Subtract the two variables.
   d. Output the result.

e. Multiply the two variables.
f. Output the result.
g. Perform integer division on the two variables (using type-casting).
h. Output the result.
i. Mod (%) the two variables (using type-casting).
j. Output the result.

10. Compile and run the program to ensure that the code is logically correct. That is, it performs the required task.

Input/Output

Standard Input

Before input can be received, memory must first be allocated to hold the input data. This is done by the variable declaration statement. Once a variable has been declared, input can be received into the program.

Create a C++ program named "`input.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Recall from earlier that the include file `iostream` allows input and output operations? In order to accept input into a program an input stream must be available. When including the file `iostream`, the `cin` (input) stream is automatically created and available for use and is associated with input via the keyboard.

Add the following code to `main`:

```
int iVar;
char cVar;
float fVar;

cout << "Enter an integer value: ";
cin >> iVar;
cout << "The integer is " << iVar << endl;

cout << "Enter a character: ";
cin >> cVar;
cout << "The character is " << cVar << endl;

cout << "Enter a floating-point value: ";
cin >> fVar;
cout << "The floating-point value is " << fVar << endl;
```

To:

- Declare three variables of type `int`, `char` and `float` then receive input into each and output their values.

The input statement, for example ,

```
cin >> iVar;,
```

Has the following characteristics:

- `>>` is the stream extraction operator and requires two operands

`cin` is always the left side operand and is the input stream.

- `iVar` is the right side operand and is the variable to receive input.

```
Compile and run the program.
=============================================================
Enter an integer value: [100]
The integer is 100

Enter a character: [f]
The character is f

Enter a floating-point value: [56.8]
The floating-point value is 56.8
=============================================================
Note:
The values encased in the [ ] is user input. This format will
follow for all examples that follow.
```

When a `cin` statement is executed the program waits for data to be entered via the keyboard and the enter key pressed.

The default behaviour of the stream extraction (>>) operator is to skip white-space characters. That is, it ignores leading white-space and uses trailing white-space to terminate each input operation. White space characters include:

- space ' '
- tab '\t'
- new-line'\n'

All input data is stored in the input buffer as a sequence of characters. That is, every key pressed on the keyboard will be stored in the input buffer including the enter/return key.

For each input operation using the operator (>>), after ignoring leading white-space, all characters will be extracted from the input buffer until either a white-space character or a character which does not match the variable type is encountered. All characters which are not extracted will remain in the input buffer and will be extracted upon the next input operation. Thus:

- If the input variable is an `int` type, only the sequence of numeral characters {0 1 2 3 4 5 6 7 8 9} are valid. Thus, if a ' '(space) or a '.' is encountered, extraction of characters will terminate and the extracted characters will be converted to their integer value and assigned to the variable. For example, if the input data is:
  - '1' will be extracted and converted to the integer value [1] then assigned to the variable.
  - "12" will be extracted and converted to the integer value [12] then assigned to the variable.

- "123" will be extracted and converted to the integer value [123] then assigned to the variable.
- '0' will be extracted and converted to the integer value [0] then assigned to the variable and ".123" will remain in the input buffer.
- "12" will be extracted and converted to the integer value [12] then assigned to the variable and the '.' will remain in input buffer.
- 1 2 '1' will be extracted and converted to the integer value [1] then assigned to the variable and " 2" will remain in the input buffer.

- If the input variable is a `char` type, any non-white-space character is valid, including the numeral characters. For example, if the input is:
  - '2' will be extracted and assigned to the variable (ascii 50).
  - a 'a' will be extracted and assigned to the variable.
  - . '.' will be extracted and assigned to the variable.
  - ~ '~' will be extracted and assigned to the variable.
  - 2abc '2' will be extracted and assigned to the variable and "abc" will remain in the input buffer.
- If the input variable is a `float/double` type, any sequence of characters which can be interpreted as a floating-point value will be extracted and converted to their floating-point value and assigned to the variable. These include the numeral characters and the character '.'. For example, if the input is:
  - '2' will be extracted and converted to the floating-point value [2.0] then assigned to the variable.
  - "2." Will be extracted and converted to the floating-point value [2.0] then assigned to the variable.
  - "2.5" will be extracted and converted to the floating-point value [2.5] then assigned to the variable.
  - 2.a "2." will be extracted and converted to the floating-point value [2.0] then assigned to the variable. "a" will remain in the input buffer.
  - 1,345.00 '1' will be extracted and converted to the floating-point value [1.0] then assigned to the variable. ",345.00" will remain in the input buffer.

The stream extraction operator (>>) can be concatenated.

Add the following code:

```
cout << "Enter the data 21.30:";
cin >> iVar >> cVar >> fVar;
cout << "iVar = " << iVar << endl;
cout << "cVar = " << cVar << endl;
cout << "fVar = " << fVar << endl;
```

To:

- Receive input into the three variables.

```
Compile and run the program.
=============================================================
Enter an integer value: [100]
The integer is 100
Enter a character: [f]
The character is f
Enter a floating-point value: [56.8]
The floating-point value is 56.8

Enter the value 21.30: [21.30]
iVar = 21
cVar = .
fVar = 30
```

During the last input operation, even there was no white-space separating the input values, the compiler extracted the data that it could match to the specific types and thus, `21` was stored `iVar`, '`.`' was stored in `cVar` and `30` was stored in `fVar`.

```
Close the file.
```

Programming Exercise:

Write a program that calculates the tax owing for a small business. The tax amount owing is 17% on net profit.

1. Create a C++ program named "`practiceExerciseInput.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Define the main function and do the following:
4. Declare `float` variables to hold the values for gross sales, gross expenses, net profit, tax owing.
5. Receive from user input the values for gross sales and gross expenses.
6. Calculate the net profit (sales less expenses).
7. Calculate the tax owing (net profit times tax percentage).
8. Output the results in the following format:

   Gross Sales: <value>
   Gross Expenses: <value>
   Net Profit: <value>
   Tax Owing: <value>

Be sure to replace <value> with the value of the respective variables. An example of code to output in a tabular format is given below:
cout << "Gross Sales: " << grossSales << endl;

11. Compile and run the program to ensure that the code is logically correct. That is, it performs the required task.

Standard Output

Output operations have previously been introduced and now will be discussed in detail. Create a file named "`output.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Notice the line:

```
#include <iostream>
```

When the file `iostream` is included in a program an output stream object named `cout` is automatically created to allow output operations to the standard output (usually screen). If the include statement is missing from a program the output stream object, `cout`, cannot be used and the program will produce an error during compilation if `cout` is accessed.

Output can be performed using the stream insertion (<<) operator which is a binary operator.

- The left operand is the output stream object.
- The right operand is the data to be output. This can be either the value of a variable or a constant, or a string of characters, referred to as a string literal.

Add the following code to `main`:

```
cout << "This is a single string output operation.";
```

To:

- Output to screen a string of text.

The statement will output a sequence of characters called a *string*. All of the characters appearing between the two quotes (") will be sent to the output buffer to be printed to

screen.

```
Compile and run the program.
=============================================================
This is a single string output operation.
```

Notice that the quotes(") do not appear in the output. This is because they have special meaning in this context. That is, the beginning and the end of a string.

If quotes are to be a part of the output string, they must be preceded with the escape character '\'. This is known as an escape sequence. Commonly used escape sequences are:

\n	New line	Cursor moves to the beginning of the next line.
\t	Tab	Cursor moves to the next tab stop.
\b	Backspace	Cursor moves to the left one space.
\r	Return	Cursor moves to beginning of current line.
\\	Backslash	Output a \
\'	Sinqle Quote	Output a '
\"	Double Quote	Output a "

Modify the statement:

```
cout << "This is a single string output operation.";
```

To the following:

```
cout << "\"This is a single string output operation.\"";
```

To:

- Output to screen a string of text that includes the quotation marks.

When the compiler encounters the escape character in a string it treats the next character with special meaning. In the case of \", it interprets the " literally and includes it in the output string.

```
Compile and run the program.
=============================================================
"This is a single string output operation."
```

Add the following code to main:

```
cout << "This is the next line of output.";
```

To:

- Output to screen another string.

```
Compile and run the program.
=============================================================
"This is a single string output operation." This is the next
line of output.
```

Notice that the output is all on the same line. This is because `cout` places all output in the output buffer one character after another. If output is to be placed on the next line, the new-line character '`\n`' must also be placed in the buffer at the appropriate place.

Modify the line of code:

```
cout << "\"This is a single string output operation.\"";
```

To the following:

```
cout << "\"This is a single string output operation.\"\n";
```

To:

- Output to screen the two strings, which will be separated by a new-line character.

Whenever the new-line character is encountered, the next output will begin on the next line.

```
Compile and run the program.
=============================================================
"This is a single string output operation."
This is the next line of output.
```

The stream insertion operator can be concatenated.

Add the following code to main.

```
cout << "\nThis output string has " << 4
     << " concatenated stream insertion operations." << endl;
```

To:

- Output to screen a concatenated string.

Another way to insert the new-line character into the output stream is to use the stream manipulator `endl`. It works by inserting a new-line character into the output buffer and then flushing the buffer. That is, send the output directly to the output device.

```
Compile and run the program.
```

```
This is a single string output operation.
This is the next line of output.
This output string has 4 concatenated stream insertion
operations.
```

```
Close the file.
```

Formatted Output

The output streamcout is defined with some default behaviour for floating-point values. Create a file named "`formattedOutput.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Add the following code to `main`:

```
cout << "Using the default behaviour: 5555.0    "  << 5555.0
      << endl;
```

To:

- Output to screen an example of the default behaviour of `cout`.

```
Compile and run the program.
```

```
Using the default behaviour: 5555.0    5555
```

Notice that the decimal point is not displayed. In order to force the decimal point, the manipulator `showpoint`, can be used.

Add the following code to `main`:

```
cout << showpoint;
cout << "Setting showpoint:           5555.0    " << 5555.0
      << endl;
```

```
Note:
If the code above does not work add the alternative code below
instead. The result is the same.
cout << setiosflags (ios::showpoint);
```

To:

- Set the stream manipulator to force the decimal point to display and output a value.

```
Compile and run the program.
=============================================================
Using the default behaviour: 5555.0     5555
Setting showpoint:           5555.0     5555.00
```

See that the decimal point is now displayed. Also, notice that there are two decimal places after the decimal point. That is because the precision is set by default to 6 digits in total.

Add the following code to `main`:

```
cout << setprecision (4);
cout << "Setting precision to 4:      5555.0     " << 5555.0
     << endl;
cout << setprecision (7);
cout << "Setting precision to 7:      5555.0     " << 5555.0
     << endl;
```

To:

- Set the precision to 4 digits and output as before, then to 7 and output.

The default behaviour `setprecision()` is to apply the precision to the number of digits displayed in total. This includes those before and after the decimal point.

```
Compile and run the program.
=============================================================
Using the default behaviour: 5555.0     5555
Setting showpoint:           5555.0     5555.00
Setting precision to 4:      5555.0     5555.
Setting precision to 7:      5555.0     5555.000
```

Thus the precision is set to print 7 digits. What if the number is larger than this?

Add the following code to main:

```
cout << "7 digit number:              5555555.     " << 5555555.
     << endl;
cout << "8 digit number:             55555555.     "
     << 55555555. << endl;
```

To:

- Output values with 7 and 8 digits to test the precision.

```
Compile and run the program.
=============================================================
Using the default behaviour: 5555.0     5555
Setting showpoint:           5555.0     5555.00
Setting precision to 4:      5555.0     5555.
Setting precision to 7:      5555.0     5555.000
7 digit number:           5555555.      5555555.
8 digit number:          55555555.      5.555556e+007
```

The seven digit number printed in full as the precision is set to 7, however, the 8 digit number is not. It is being displayed in scientific notation, which is the default, and occurs when the number is too large for the precision value. In order to change this to fixed decimal notation, the `fixed` manipulator is used.

Add the following code to main:

```
cout << fixed;
cout << "Setting fixed:            55555555.     "
     << 55555555. << endl;
```

To:

- Set the output to fixed decimal notation and output a value.

```
Compile and run the program.
=============================================================
Using the default behaviour: 5555.0     5555
Setting showpoint:           5555.0     5555.00
Setting precision to 4:      5555.0     5555.
Setting precision to 7:      5555.0     5555.000
7 digit number:           5555555.      5555555.
8 digit number:          55555555.      5.555556e+007
Setting fixed:           55555555.      55555555.0000000
```

Notice now that the notation is in fixed decimal place instead of scientific, however, the precision now applies to the digits after the decimal place. This is what happens when fixed is used. The precision now applies to the places after the decimal point.

```
Important Note: Once a format flag has been set, it remains
set until explicitly re-set/un-set.
```

So far, the output has been formatted into columns by hard-coding the spaces. This can also be achieved by using the `setw()` function which takes an integer parameter that is the width of the output column.

- The `setw()` function sets the width of the *next* output item *only*. That is, outputs the data in a column that is the specified value characters wide.

- If the size of the data is less than the value given, the remaining spaces are filled with blank spaces.
- If the size of the data is larger than the value given, the specified size is over-ridden by the actual size.
- The function `setw()` *only* applies to the next output item.

Modify the current code within `main` to the following:

```
cout << setw (30) <<"Using the default behaviour:"
     << setw (12) << "5555.0"
     << setw (10) <<5555.0 << endl;

cout << showpoint;
cout << setw (30) <<"Setting showpoint:"
     << setw (12) << "5555.0"
     << setw (10) <<5555.0 << endl;

cout << setprecision (4);
cout << setw (30) << "Setting precision to 4:"
     << setw (12) << "5555.0"
     << setw (10) << 5555.0 << endl;

cout << setprecision (7);
cout << setw (30) << "Setting precision to 7:"
     << setw (12) << "5555.0"
     << setw (10) <<5555.0 << endl;

cout << setw (30) << "7 digit number:"
     << setw (12) << "5555555."
     << setw (10) <<5555555. << endl;
cout << setw (30) << "8 digit number:"
     << setw (12) << "55555555."
     << setw (10) <<55555555. << endl;

cout << fixed;
cout << setw (30) << "Setting fixed:"
     << setw (12) << "55555555."
     << setw (10) <<55555555. << endl;
```

To:

- Set the width of each of the output data and output accordingly.

```
Compile and run the program.
=============================================================
 Using the default behaviour:       5555.0       5555
          Setting showpoint:        5555.0    5555.00
     Setting precision to 4:       5555.0      5555.
      Setting precision to 7:       5555.0   5555.000
               7 digit number:    5555555.   5555555.
               8 digit number:   55555555.5.555556e+007
```

```
Setting fixed:    55555555.55555555.0000000
```

Notice that the data does not seem aligned correctly. It would be *better* if the data was left justified.

Add the following code before the `cout` statements added above.

```
cout << left;
```

To:

- Format the output to the left of the field.

```
Compile and run the program.
=============================================================
Using the default behaviour:  5555.0       5555
Setting showpoint:            5555.0       5555.00
Setting precision to 4:       5555.0       5555.
Setting precision to 7:       5555.0       5555.000
7 digit number:               5555555.     5555555.
8 digit number:               55555555.    5.555556e+007
Setting fixed:                55555555.    55555555.0000000
```

When using `setw()`, the extra spaces are filled by default with a space. It can, however, be filled with any character at all.

Add the following code to `main` before all the current code:

```
cout << setfill ('_');
```

```
Compile and run the program.
=============================================================
Using the default behaviour:__5555.0_____5555_____
Setting showpoint:___________5555.0_____5555.00___
Setting precision to 4:______5555.0_____5555._____
Setting precision to 7:______5555.0_____5555.000__
7 digit number:______________5555555.___5555555.__
8 digit number:______________55555555.___5.555556e+007
Setting fixed:_______________55555555.___55555555.0000000
```

The blank spaces previously are now filled with the character '_'. This will remain so until changed. Usually, it will be re-set to the default character ' ' (space).

```
Close the file.
```

Programming Exercise:

Extend the previous program so that the output is formatted using the stream manipulators and/or format functions.

1. Create a C++ program named "`practiceExerciseOutput.cpp`".
2. Copy the code from "`practiceExerciseInput.cpp`" into this file.
3. Include the header file `iomanip`.
4. Alter the code that prints the tabular results to the following:

   Gross Sales:.....$<value>
   Gross Expenses:.....$<value>
   Net Profit:.....$<value>
   Tax Owing:.....$<value>

   * <value> should be replaced by the values of the respective variables,
   * Format the output to two decimal places. i.e. `3500.50` or `2500.00`
   * Use the `setw()` function to set the width of each output item.
   * The "...." should be filled using the `setifll()` function.
5. Compile and run the program to ensure that the code is logically correct. That is, it performs the required task.

Control Structures

Code can be executed in one of three ways:

- Sequentially - Statement(s) are executed one after another in a top-down approach.
- Selectively - Statement(s) are executed by making a choice. Do this, OR do that.
- Repetitively - Statement(s) are executed repeatedly.

Thus far, the code has been sequential. However, at times the programmer may require certain code to be executed based upon some certain condition being met. Other times, the certain code will be repeatedly executed. Two specific control structures are used to implement selection and repetition.

Relational Operations

The condition upon which a selection or repetition structure is based, most often is the result of a relational operation. The relational operators are:

==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Each of the relational operators is a binary operator. When used in an expression, the result of a relational operation is either 1 (true) or 0 (false). Either the condition is true, or it is false.

Create a file named "`relationalOperations.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Add the following code to `main`:

```
cout << "1 < 2       := " << (1 < 2) << endl;
cout << "1 > 2       := " << (1 > 2) << endl;
cout << "1 == 2      := " << (1 == 2) << endl;
cout << "1 != 2      := " << (1 != 2) << endl;
-------------------------------------------------------------
Note:
The () are required around the relational expression inside of a
cout statement.
```

To:

- Test the relational operators on integer operands and output the results.

```
Compile and run the program.
=============================================================
1 < 2       := 1
1 > 2       := 0
1 == 2      := 0
1 != 2      := 1
```

The code above uses the relational operators with int types. Relational operators also work any of the built-in types and the operands can be of mixed types.

Add the following code to `main`:

```
cout << "1.0 < 2.0   := " << (1.0 < 2.0) << endl;
cout << "1.0 > 2.0   := " << (1.0 > 2.0) << endl;
cout << "1.0 == 2    := " << (1.0 == 2) << endl;
cout << "1.0 != 2    := " << (1.0 != 2) << endl;
cout << "'a' < 'b'   := " << ('a' < 'b') <<endl;
cout << "'A' == 65   := " << ('A' == 96) << endl;
```

To:

- Test the relational operators on floating-point, chararacter and mixed-type operands and output the results.

```
Compile and run the program.
=============================================================
1 < 2       := 1
1 > 2       := 0
1 == 2      := 0
1 != 2      := 1
1.0 < 2.0   := 1
1.0 > 2.0   := 0
1.0 == 2    := 0
1.0 != 2    := 1
'a' < 'b'   := 1
'A' == 65   := 1
```

In an evaluation with a `char` type the ascii value of the character is used. The result of a relational expression is either 1 (true) or 0 (false).

```
Close the file.
```

Selection - `if` Statement

The structure provided by C++ for selection is the `if` statement. Create a file named "`selection.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Add the following code to `main`:

```
int input1, input2;

cout << "Enter two integer values: ";
cin >> input1 >> input2;

if (input1 != input2)
   cout << "The two values entered are different!" << endl;
```

To:

- Declare two `int` and receive input into them, then test for in-equality and output accordingly.

```
Compile and run the program.
═══════════════════════════════════════════════════════
Enter two integer values: [123 456]
The two values entered are different!

Run the program to produce the output:
Enter two integer values: [123 123]
═══════════════════════════════════════════════════════
Note:
The second run of the program produces no output.
```

Currently, the program executes code *only* if the condition is *true*. It is possible to have code executed if the condition is *false*. That is, if the condition is *not true*. Modify the `if` statement in `main` to:

```
if (input1 != input2)
   cout << "The two values entered are different!" << endl;
```

```
else
   cout << "The two values entered are the same!" << endl;
```

To:

- Test if `input1` is *not equal* to `input2` and if true print a message indicating that they are different.
- Otherwise, print a message indicating that they are the same.

```
Compile and run the program.
=============================================================
Enter two integer values: [123 123]
The two values entered are the same!
```

The above code allows for only two possible conditions. Either the values are the same or they are not (i.e. different). The statement can be restructured to allow for a third, or even more, conditions to be tested.

Modify the `if else` statement in `main` to:

```
if (input1 < input2)
   cout << input1 << " is less than " << input2 << endl;
else if ( input1 > input2)
   cout << input1 << " is greater than " << input2 << endl;
else
   cout << input1 << " is equal to " << input2 << endl;

-------------------------------------------------------------
Note:
These are cascaded if statements. I.e. one after the other, all
logically linked.
```

To:

- Test if `input1` is *less than* to `input2` and if true print a message indicating so.
- Otherwise test if `input 1` is *greater than* `input2` and if true print a message indicating so.
- Otherwise (`input1` must be *equal to* `input 2`) print a message indicating that `input1` is equal to `input2`

```
Compile and run the program.
=============================================================
Enter two integer values: [123 125]
123 is less than 125

Run the program to produce the output:
Enter two integer values: [123 54]
123 is greater than 54

Run the program to produce the output:
Enter two integer values: [124 124]
```

```
124 is equal to 124
```

Logical expressions can be concatenated using the logical OR (`||`) or AND (`&&`) operators.

Modify the cascaded `if` statement to:

```
if (input1 > 0 && input2 > 0)
{
   if (input1 < input2)
      cout << input1 << " is less than " << input2 << endl;
   else if ( input1 > input2)
      cout << input1 << " is greater than " << input2 << endl;
   else
      cout << input1 << " is equal to " << input2 << endl;
}
else
   cout << "Input must be > 0! " << endl;
```

```
Note:
These are nested and cascaded if statements. I.e. one inside
another and cascaded.
```

To:

- Test if `input1` is *greater than* 0 **AND** `input2` is *greater than* 0 and if true execute the code inside of the `{}`.

Consider the expression:

```
(input1 > 0 && input2 > 0)
```

In order for the expression to be evaluated as true, both expressions on each side of the `&&` must be true. That is, `input1` must be *greater than* 0 **and** `input2` must be *greater than* 0. If *either* side is false, the entire expression is false.

```
Compile and run the program.
```

```
Enter two integer values: [0 123]
Input must be > 0!
```

Alternatively, the logical operator (`||`) can be used.

Modify the nested and cascaded `if` statements to:

```
if (input1 > 0 || input2 > 0)
{
   if (input1 < input2)
      cout << input1 << " is less than " << input2 << endl;
   else if ( input1 > input2)
      cout << input1 << " is greater than " << input2 << endl;
   else
      cout << input1 << " is equal to " << input2 << endl;
```

```
}
else
   cout << "Input must be > 0! " << endl;
```

To:

- Test if `input1` is *greater than* 0 **OR** if `input2` is *greater than* 0 and if either is true the code inside the `{}` will be executed.

Consider the expression:

```
(input1 > 0 || input2 > 0)
```

In order for it to be evaluated as true, *only one* expression on either side of the `||` must be true. That is, `input1` must be *greater than* 0 **OR** `input2` must be *greater than* 0. If *both* sides are false, the entire expression is false.

```
Compile and run the program.
=============================================================
Enter two integer values: [0 123]
0 is less than 123

-------------------------------------------------------------
Enter two integer values: [123 0]
123 is greater than 0

-------------------------------------------------------------
Enter two integer values: [0 0]
Input must be > 0!
```

As long as the condition of an `if` statement evaluates to an integral value, the expression will evaluate to true (1) if the value is non-zero, otherwise it will be false (0). Modify the nested and cascaded `if` statements to:

```
if (input1)
{
   if (input2)
   {
      if (input1 < input2)
         cout << input1 << " is less than " << input2
              << endl;
      else if ( input1 > input2)
         cout << input1 << " is greater than " << input2
              << endl;
      else
         cout << input1 << " is equal to " << input2
              << endl;
   }
   else
      cout << "Input2 must not be 0!" << endl;
}
else
```

```
cout << "Input1 must not be 0! " << endl;
```

To:

- Test if `input1` is *true* and if true execute the code inside of the `{}` otherwise, print a message indicating that `input1` must not be 0.
- Test if `input2` is *true* and if true execute the code inside of the `{}` otherwise print a message indicating that `input2` must not be 0.

```
Compile and run the program.
=============================================================
Enter two integer values: [0 454]
Input1 must not be 0!

-------------------------------------------------------------
Enter two integer values: [454 0]
Input2 must not be 0!

-------------------------------------------------------------
Enter two integer values: [0 0]
Input1 must not be 0!

-------------------------------------------------------------
Enter two integer values: [124 567]
124 is less than 567

-------------------------------------------------------------
Enter two integer values: [-123 -456]
-123 is greater than -456
```

Short-circuit Evaluation

C++ evaluates logical expressions based upon a process called short-circuit evaluation. That is, if the result of a logical expression can be determined before evaluating the entire expression, the evaluation will stop. For example, consider the expression:

```
(input1 > 0 && input2 > 0)
```

If

```
input1 > 0
```

evaluates to *false* then the other side of the expression

```
(input2>0)
```

does *not* need to be evaluated. This is so because regardless of whether the right expression is true or false the result of the entire expression will still be false. If either side of `&&` is false, the result is false.

Likewise, in the expression:

```
(input1 > 0 || input2 > 0)
```

If

```
input1 > 0
```

evaluates to *true*, then evaluation stops as the entire expression will be evaluated to true. This is similar to use of `&&`, however, with `||` only one side of the expression must be true for the entire expression to be true.

This has implications if a simple expression has side-effects. For example, consider the code:

```
(input1++ > 0 || input2++ > 0)
```

In this code, the variables `input1` and `input2` are being incremented after being used in the evaluation. If the left side of the `||` expression is true, the right side will not be executed. Hence, the variable `input2` will never be incremented. Thus:

Be careful about having code that has side-effects in logical expressions.

Logical expressions are evaluated left to right and `&&` has a higher precedence than `||` which has a higher precedence than the relational operators [< <= > >= == !=].

```
Close the file.
```

Programming Exercise:

Extend the program from the previous example, to validate the values input by the user and only continue if the values are valid by using if-else statements

1. Create a C++ program named "`practiceExerciseIf.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Copy the code from the file "`practiceExerciseOutput.cpp`".
4. Modify the code by adding an if statement after each input item and testing that the values input are valid 0. If they are invalid then calculation cannot continue and an appropriate error message should be output and the program must terminate. The pseudo-code below illustrates this.

```
INPUT gross_sales
IF gross_sales > 0
    INPUT gross_expenses
    IF gross_expenses >= 0
        net_ profit = gross_sales – gross_expenses
        Tax = net_profit * tax_percentage
```

```
            OUTPUT // the tabular results here
        ELSE
            OUTPUT "Can't continue, expenses can't be < 0
        END IF
    ELSE
        OUTPUT "Can't continue, sales can't be <= 0
    END IF
```

5. Compile and run the program to ensure that the code is logically correct.

Selection - `switch` Statement

Another selection structure is the `switch` statement which implements multiple selection cases. The primary difference between an `if` statement and a `switch` statement is that `if` statements readily deal with ranges of values whereas `switch` statements do not. Also, the expression is evaluated to an integral value such as: `int`, `char`, `bool` types or the result of a logical expression, 1(true)/0(false).

Create a file named "`switch.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Add the following code to main:

```
int i;
cout << "Enter a number: ";
cin >> i;

switch (i % 2)
{
   case 0:     cout <<"The number " << i << " is even!" << endl;
   case 1:     cout << "The number " << i << " is odd!" << endl;
}
```

```
Compile and run the program.
```

```
Enter a number: [101]
The number 101 is odd!

Enter a number: [100]
The number 100 is even!
The number 100 is odd!
```

Notice that the second run of the program does not produce the desired result. This is because the `switch` statement works by locating the appropriate `case` value, then executing all code from that point until either it encounters a `break` statement or the end of the `switch`. Modify the switch statement to be:

```
switch (i % 2)
{
   case 0:     cout <<"The number " << i << " is even!" << endl;
               break;
   case 1:     cout << "The number " << i << " is odd!" << endl;
               break;
}
```

```
Compile and run the program.
Enter a number: [101]
The number 101 is odd!

Enter a number: [100]
The number 100 is even!
```

Case values can be of `char` types. Comment out the current code in main (to avoid unnecessary repetition during execution) and add the following code:

```
char ch;
cout << "Enter (a) or (b) or (c): ";
cin >> ch;

switch (ch)
{
   case 'a':   cout <<"" << endl;
               break;
   case 'b':   cout <<"" << endl;
```

```
        break;
   case 'c':   cout <<"" << endl;
        break;
}
```

```
Compile and run the program.
============================================================
Enter (a) or (b) or (c): [a]
a was input.

------------------------------------------------------------
Enter (a) or (b) or (c): [b]
b was input.

------------------------------------------------------------
Enter (a) or (b) or (c): [c]
c was input.
------------------------------------------------------------

Enter (a) or (b) or (c): [z]
```

Notice on the last run that there was no output. Often when switch value is not one of the cases, a default case is given.

Modify the switch statement to:

```
switch (ch)
{
      case 'a': cout << ch << " was input." << endl;
           break;
      case 'b': cout << ch << " was input." << endl;
           break;
      case 'c': cout << ch << " was input." << endl;
           break;
      default: cout << "Invalid input!"<<endl;
}
```

```
Compile and run the program.
============================================================
Enter (a) or (b) or (c): [z]
Invalid input!

------------------------------------------------------------
Enter (a) or (b) or (c): [A]
Invalid input!
```

The switch will now deal with all possible cases. However, if the user enters an upper case value for a, b or c, these will be deemed invalid.

Modify the switch statement to:

```
switch (ch)
{
      case 'a': case 'A':
              cout << ch << " was input." << endl;
              break;
      case 'b': case 'B':
              cout << ch << " was input." << endl;
              break;
      case 'c': case 'C':
              cout << ch << " was input." << endl;
              break;
   default:   cout << "Invalid input!"<<endl;
}
```

```
Compile and run the program.
=============================================================
Enter (a) or (b) or (c): [A]
A was input.
-------------------------------------------------------------
Enter (a) or (b) or (c): [a]
a was input.
```

```
Close the file.
```

Programming Exercise:

Extend the program from the previous example, to receive an additional input value that indicates the tax rate and uses a switch statement to calculate the correct tax owing.

1. Create a C++ program named "`practiceExerciseSwitch.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Copy the code from the file "`practiceExerciseIf.cpp`".
4. Add code to receive from user input a character value which indicates the tax rate based on the table below.

   a = 17%
   b = 20%
   c = 25%

   i.e. if the user enters 'c', the tax rate is 25%
   if the user enters 'z' the base rate of 17% is used.

5. Replace the code that calculates the tax owing with a switch statement and calculate the tax based on the values entered above e.g. a, b, c. I.e. Place the current code inside the switch statement.
6. Compile and run the program to ensure that the code is logically correct.

Value Range

Switch statements are not suited for a large range of values, Ie. 0-100. In order to do so, each of the cases from 0 to 100 must be included I.e.

```
case 0:
case 1:
...
case 100: statement(s); break;
```

This would result in unnecessarily *bloating* the code when a simple `if` statement could achieve the same result with much less code.

Repetition – `for` Statement

When the same code segment needs to be executed several times, for example, find the average of *n* numbers input by the user, a repetition structure can be used rather having *n* input statements. Repetition is also known as looping.

Create a file named "`forLoop.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Add the following code to main:

```
int num, total = 0;

cout << "Enter a number: ";
cin >> num;
total += num;

cout << "Enter a number: ";
```

```
cin >> num;
total += num;

cout << "Enter a number: ";
cin >> num;
total += num;

cout << "Enter a number: ";
cin >> num;
total += num;

cout << "Enter a number: ";
cin >> num;
total += num;

cout << "The average is: " << total / 5.0 << endl;
```

To:

- Receive input for 5 values, add the value to `total`, then calculate and print the average.

```
Compile and run the program.
============================================================
Enter a number: [12]
Enter a number: [56]
Enter a number: [45]
Enter a number: [87]
Enter a number: [99]
The average is: 59.8
```

This code does adequately perform the job, however, the following code is repeated several times.

```
cout << "Enter a number: ";
cin >> num;
total += num;
```

If the program was to average 100 numbers, for example, the code would soon become cumbersome. In this case, the code that is repeated can be placed inside a repetition structure. The `for` loop is one such structure.

Modify `main` to the following code:

```
int num, total = 0;

for (int i = 0; i < 5; i++)
{
   cout << "Enter a number: ";
   cin >> num;
   total += num;
```

```
}
cout << "The average is: " << total / 5.0 << endl;
```

The three statements inside of the `for` loop `()` are:

```
int i = 0
```

This is the initialization statement. It is used to initialize the variable(s) controlling the loop. It is executed once.

```
i < 5
```

This is the condition which controls the loop. If and then while `i` is less than 5 the loop will continue to iterate. It is first executed after the initialization statement, then again after each update statement.

```
i++
```

This is the update condition. It is executed each time after the code inside the loop executes.

```
Compile and run the program.
═══════════════════════════════════════════════════════════
Enter a number: [12]
Enter a number: [56]
Enter a number: [45]
Enter a number: [87]
Enter a number: [99]
The average is: 59.8
```

The code is now more compact and achieves the same result. Also, should the program need to be modified to find the average of 100 numbers, the change to the program is minimal. Merely replace the code:

```
i < 5
```

And

```
total / 5.0
```

With:

```
i < 100
```

And

```
total / 100.0
```

Alternatively, the number of iterations can be determined by user input.

Modify `main` to the following:

```
int num, total = 0, size;
cout <<"How many numbers will be input: ";
```

```
cin >> size;

for (int i = 0; i < size; i++)
{
   cout << "Enter a number: ";
   cin >> num;
   total += num;
}
cout << "The average is: "
      << static_cast<double>(total) / size << endl;
```

To:

- Ask the user for input for the number of numbers to receive then input the numbers then calculate and print the average.

The program now works for however many values the user enters. By coding this way the program remains un-bloated and is flexible. It deals with 1 or 1000 numbers and the size of the code remains the same.

```
Close the file.
```

Programming Exercise:
Write a program to receive N input items from the user and calculate the largest, smallest, and average value input.

1. Create a C++ program named "`practiceExerciseFor.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Declare an int variable to hold the number of values to be input.
4. Declare an int variable to hold the current number input.
5. Declare an int variable to hold the sum of all values entered.
6. Declare an int variable to hold the largest value input.
7. Declare an int variable to hold the smallest value input.
8. Declare a float variable to hold the average value entered.
9. Ask the user for and receive input for the number of items to enter.
10. Using a for loop iterate N times,
    *Receive input for the current number.
    *Set the largest number by the following pseudo-code
      IF current_number is greater than largest
            largest = current_number
    *Set the smallest number by the following pseudo-code
      IF current_number is smaller than smallest
            smallest = current_number
    *Add the current number to the sum

11. Calculate the average number by dividing the sum by the N (you may need to use static_cast<>())
12. Output the values for largest, smallest and average.
13. Compile and run the program to ensure that the code is logically correct.

Repetition – `while` Statement

Another repetition structure is the `while` statement. It too will iterate until a certain pre-defined condition becomes true.

Create a program named "`whileLoop.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Add the following code to main:

```
int num = 0, count = 0, total = 0;

cout << "Enter a (-1 to finish): ";
cin >> num;

while (num != -1)
{
   total += num;
   count++;
   cout << "Enter a number (-1 to finish): ";
   cin >> num;
}
cout << "The average is: "
      << static_cast<double>(total) / count << endl;
```

To:

- Iterate, getting input and adding to `total` until the user enters -1, then calculate and print the average.

This loop will iterate until the user enters the value `-1`, upon which time it will proceed to the next statement immediately following the loop.

```
Compile and run the program.
```

```
Enter a number: [10]
Enter a number: [15]
Enter a number: [20]
Enter a number: [-1]
The average is: 15
```

Note: The fundamentals of the `while` statement are similar to that of the `for` statement and they can in-fact be used interchangeably. For example:

1. Declare a variable to control the loop *before* the while loop.
   I.e. `int i = 0;`
2. Place the control condition inside the `()`.
   I.e. `while (i < 5)`
3. Add the update statement at the *end* of the loop.
   I.e. `i++;`

However, the general motive to choose a `while` loop over a `for` loop is when the number of iterations of the loop is dependent upon something which happens inside of the loop. I.e. The user enters a value to indicate termination of the loop.

```
Close the file.
```

Programming Exercise:

Extend the previous exercise to use a while loop instead of a for loop and rather than iterating N times, input will stop when the user enters the value -9999.

1. Create a C++ program named "`practiceExerciseWhile.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Copy the code from "`practiceExerciseFor.cpp`".
4. Modify the for loop to a while loop that iterates until the value entered is -9999. Do not use the value -9999 in the smallest, largest, etc.
5. The rest of the code remains the same.
6. Output the results.
7. Compile and run the program to ensure that the code is logically correct.

Repetition – `do while` Statement

The third repetition structure is the `do while` loop. It behaves almost identically to the `while` loop except that the code is *always* executed *before* the condition is tested. Hence, the code will always be executed at least one time.

Create a program named "dowhileLoop.cpp" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Add the following code to `main`:

```
int num = 0, count = 0, total = 0;
bool validInput = false;

do
{
   cout << "Enter a number (-1 to finish): ";
   cin >> num;
   if (cin.fail ())
   {
      validInput = false;
      cin.clear ();
      cin.ignore (100, '\n');
   }
   else
   {
      validInput = true;
      if (num != -1)
      {
         total += num;
         count++;
      }
   }
} while (!validInput || num != -1);

cout << "The average is: "
      << static_cast<double>(total) / count << endl;
```

To:

- Place the input inside a do-while loop that will iterate if the input is invalid, or if the input does not equal the sentinel value '-1', which indicates to terminate the loop.

The code is similar to that in "`while.cpp`", however, all the code is 'wrapped' inside the do-while statement and data validation is included. Consider the following code:

```
cin >> num;
if (cin.fail ())
```

```
{
   validInput = false;
   cin.clear ();
   cin.ignore (100, '\n');
}
```

It checks the input stream after an input operation to see if it has failed due to invalid data being input for the required type. i.e. 'a' received for `int` type. If this is the case, the stream must be cleared of its 'bad' state and the characters in the input stream discarded so that further input operations can proceed.

Typically this code is put inside a do-while statement to continue getting input until the data is valid.

```
Compile and run the program.
```

```
Enter a number (-1 to finish): [1]
Enter a number (-1 to finish): [10]
Enter a number (-1 to finish): [b]
Invalid input!

Enter a number (-1 to finish): [15]
Enter a number (-1 to finish): [.]
Invalid input!

Enter a number (-1 to finish): [20]
Enter a number (-1 to finish): [aaaaaa]
Invalid input!

Enter a number (-1 to finish): [-1]
The average is: 11.5
```

```
Close the file.
```

Programming Exercise:

Extend the previous exercise to use a do-while loop instead of a while loop. Input will stop when the user enters the value -9999 or when 5 invalid values are entered. An invalid value is a value less than 0.

1. Create a C++ program named "`practiceExerciseDoWhile.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Copy the code from "`practiceExerciseWhile.cpp`".

4. Declare an int variable to store the number of invalid data that has been entered and initialize it to 0. This value should be incremented each time an invalid value has been entered.
5. Modify the while loop to a do-while loop. The loop will terminate when the user enters -9999 OR when the user enters 5 invalid values.
6. Don't use the invalid values or the value -9999 in the largest, smallest, etc.
7. The rest of the code remains the same.
8. Output the results.
9. Compile and run the program to ensure that the code is logically correct.

User-Defined Functions

All C++ programs are constructed of one or more *user-defined* functions of which one must be `main`. The programmer may also define additional functions. The primary motive for implementing user-defined functions is to ensure that the program is truly modular. That is, it consists of modules (functions) as it is considered a good programming practice to break the larger programming task into several smaller tasks. Each of these tasks will be defined within a separate function.

Another reason for implementing functions is to restrict the amount of duplicate code. For example, if a program receives several integer inputs, rather than having the bulky code to validate input several times throughout the program, one instance can be placed inside a function and that function can be invoked several times.

Create a program named "`functionVoid.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

There are three components to user-defined functions.

Prototype

The prototype is a declaration to the compiler that a function which has certain characteristics (return type, name and parameters) is defined later. The compiler needs to be aware of the functions existence before it encounters a function call. The prototype is typically placed *before* the main function. I.e. Before any reference to the function occurs.

The three distinct parts of a function prototype are:

1. Return type: The *type* of the value being returned by the function, or the `void` type if none.
2. Function name: The function identifier. The same naming rules apply as for variable or constant identifiers.
3. Parameter list: Zero or more parameters (data) passed into the function. The list of parameters, if any, are placed between the `()`.

Add the following code *before* main:

```
void celcius2fahrenheit ();
```

Note:
Don't forget the semi-colon ';' at the end of the prototype.

To:

- Declare a function named `celcius2fahrenheit()` which has a return type of `void` and takes no parameters (i.e. no data passed to function).

Definition

The function definition contains the code which will be executed each time the function is invoked (called). The *header* of the definition *looks* the same as the function prototype. Additionally, the definition contains the block of code to be executed which is contained within the `{}`. Typically, the definition is placed *after* the `main` function.

Add the following code *after* `main`:

```
void celcius2fahrenheit () //<-- this is called the header
{
   float celcius;
   cout << "Enter temperature in degrees Celcius: ";
   cin >> celcius;

   cout << "is equal to " << ((9/5.0) * celcius + 32)
      << " degrees Fahrenheit.\n";
}
```

To:

- Define the function `celcius2fahrenheit()`.

Notice the absence of a '`;`' at the end of the function header. Unlike the prototype, there is no semi-colon ';' there as it would alter the meaning from a function definition to a function prototype.

Compile and run the program.

Note: There will not be any output.

Currently there is no output as `main` contains no executable code except `return 0;`. In order for the code within the function to be executed, the function must be invoked (called).

Call

The function call is an instruction to the compiler to invoke the function and subsequently execute the code within the function definition.

Add the following code to `main`:

```
celcius2fahrenheit ();
```

To:

- Call (invoke) the function `celcius2fahrenheit()`.

Whenever the function call, `celcius2fahrenheit ()`, is encountered execution will *jump* to the function definition and the contained code executed. When the end of the function is reached, the function is exited and execution continues with the next statement *after* the function call.

```
Compile and run the program.
```

```
Enter temperature in degrees celcius: [23]
is equal to 73.4 degrees Fahrenheit.
```

The function `celcius2fahrenheit()` can be called any number of times and from within any user-defined function.

Return Value

A function may return a value to the calling function by way of the `return` statement. The value returned may be a variable (e.g. `fahr`), a simple expression (e.g. 0, 4.5 or `true`) or a complex expression (e.g. 4.5 * 45 / `fahr`).

Recall that the `main` function has as its last statement `return 0;`. This is the value that it returns to the operating system each time it finishes execution. Main *always* returns a value.

- Other functions may or may not return a value. If a function does not return a value it must still declare a return *type*, in this case, `void` (i.e. nothing).

```
Save the current file as "functionNonVoid.cpp".
```

Add the following function prototypes *before* `main`.

```
float fahrenheit2celcius ();
```

To:

- Declare the function `fahrenheit2celcius()`

Add the following function definition *after* `main`.

```
float fahrenheit2celcius ()
{
   float fahr;
   cout << "\nEnter temperature in degrees Fahrenheit: ";
   cin >> fahr;
```

```
    return ((5/9.0) * (fahr - 32));
}
```

To:

- Define the function `fahrenheit2celcius()`.

In a non-void function, a return statement ***must*** be given and only one value can be returned. It is the value of the expression or variable that is returned. The type returned should be the same as the type declared, however, the compiler will do type conversions where possible.

There are several ways to use the value returned from a function, for example:

- Assign the value to a variable.
- Directly in an output statement as an operand.
- As a parameter to another function.
- As part of a conditional expression.

Add the following code to `main`:

```
float celc = 0;

celc = fahrenheit2celcius ();
cout << "is equal to " << celc
     << " degrees Celcius. " << endl;
```

To:

- Call the function `fahrenheit2celcius()` and assign the returned value to the variable and output the results.

Values returned by a function should be used by the calling function accordingly. In this example the value returned by the function, `fahrenheit2celcius()`, is assigned to the variable `celc` and used in the output statement.

```
Compile and run the program.
```

```
Enter temperature in degrees Celcius: [23.89]
is equal to 75.002 degrees Fahrenheit.

Enter temperature in degrees Fahrenheit: [75.002]
is equal to 23.89 degrees Celcius.
```

Parameters - Pass by Value

Generally, a function whose task is to perform some calculation and return the result will very often not have input statements inside the function itself. Instead, the input will be received elsewhere and passed to the function. Alternatively, the input may not come from user input at all, for example, the input may come from a file.

By passing the data to the function, it becomes more general and can be used by various code. The data received by a function upon invocation is called a parameter and can be any built-in or user-defined type. A function may have zero or more parameters.

```
Remove the function prototype, definition and call to
celcius2fahrenheit() and save the file as
"functionPassByVal.cpp".
```

Modify the prototype `fahrenheit2celcius()` to the following.

```
float fahrenheit2celcius (float);
```

To:

- Declare the function `fahrenheit2celcius()` with one parameter of type `float`.

The name of a parameter can be omitted in the prototype or it can differ from that in the definition. In the prototype, all the compiler needs to know is the return type, the function name and the number and type of parameters, it ignores the parameter name(s).

Modify the definition for `fahrenheit2celcius()` to the following:

```
float fahrenheit2celcius (float fahr)
{
   return ((5/9.0) * (fahr - 32));
}
```

To:

- Define the function `fahrenheit2celcius()` which takes one `float` parameter.

In the function definition, the parameter declarations are in effect variable declarations. In the function `fahrenheit2celcius()` the parameter `fahr` is a variable whose scope is restricted to that function. That is, it is accessible only within the function itself. Also, it is a *special* variable in that its initial value is given when the function is called.

Modify `main` to:

```
float fahrenheit;
cout << "\nEnter temperature in Fahrenheit: ";
cin >> fahrenheit;

cout << "is equal to  "
     << fahrenheit2celcius (fahrenheit)
     << " degrees Celcius.\n";
```

To:

- Receive user input for the degrees in Fahrenheit and output the result, which includes a function call to `fahrenheit2celcius(fahrenheit)`.

When passing variables to functions, only the variable's name is given (i.e. do **not** include the type.). The data given in the function call is called the *argument*.

In the function call, `fahrenheit2celcius(fahrenheit)`, the returned value is being used as an operand in the `cout` statement. After the statement terminates the value is no longer accessible.

```
Compile and run the program.
```

```
Enter temperature in Fahrenheit: [113]
is equal to 45 degrees Celcius.
```

Arguments to *pass by value* functions can also be un-named constant values such as `145.89`.

Add the following code to `main`:

```
cout << "\nTemperature Fahrenheit: " << 145.89
   << "\nis equal to " << fahrenheit2celcius (145.89)
   << " degrees Celcius.\n";
```

To:

- Call the function `fahrenheit2celcius()` with a hard-coded `double` value.

It is important to note that the number and type of the argument(s) *must* match that of the parameter(s). The compiler will, however, perform implicit type conversions where possible.

Add the following code to `main`:

```
cout << "\nTemperature Fahrenheit: " << 145
   << "\nis equal to " << fahrenheit2celcius (145)
   << " degrees Celcius.\n";
```

To:

- Call the function `fahrenheit2celcius()` with a hard-coded `int` value.

```
Compile and run the program.
```

```
Enter temperature in Fahrenheit: [113]
is equal to 45 degrees Celcius.

Temperature Fahrenheit: 145.89
is equal to 63.2722 degrees Celcius

Temperature Fahrenheit: 145
is equal to 62.7778 degrees Celcius
```

The compiler converted `145` to `145.0` when assigning the value to the parameter.

Add the following code to `main`:

```
cout << "\nTemperature Fahrenheit: " << 'A'
   << "\nis equal to " << fahrenheit2celcius ('A')
   << " degrees Celcius\n";
```

To:

- Call the function `fahrenheit2celcius()` with a hard-coded `char` value.

Again, the compiler performed an implicit type conversion by converting the ascii value of 'A' (65) to a `float` (65.0) for the assignment to the parameter variable.

```
Compile and run the program.
```

```
Enter temperature in Fahrenheit: [65]
is equal to  18.3333 degrees Celcius.

Temperature Fahrenheit: 145.89
is equal to 63.2722 degrees Celcius.

Temperature Fahrenheit: 145
is equal to 62.7778 degrees Celcius.

Temperature Fahrenheit: A
is equal to 18.3333 degrees Celcius.
```

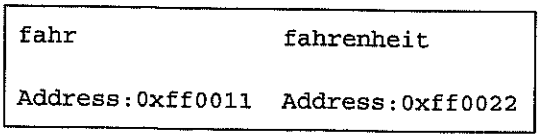
```
Close the file.
```

Parameters - Pass by Reference

The parameters seen so far are called ***pass by value*** parameters. That is, the *value* of the argument is passed to the function and assigned to the parameter variable.

Upon invocation the argument and the parameter have the same value, however, they are completely separate entities residing in different memory locations. By the time the function terminates, the value of the parameter variable may have changed but this is not reflected in the argument's value as they are not the same variable. This is the default behaviour of parameter passing. For example:

Pass by Value:



Create a file named "`functionPassByRef.cpp`" and add the following code:

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

int main ()
```

```
{
   return 0;
}
```

To:

- Define the `main` function.

Add the following function prototype.

```
void swap (char ch1, char ch2);
```

To:

- Declare a function which takes two parameters and returns no value.

Add the following function definition.

```
void swap (char ch1, char ch2)
{
   char tmp = ch1;
   ch1 = ch2;
   ch2 = tmp;
}
```

To:

- Define the function `swap()` which will swap the values of the two parameter variables.

Add the following code to `main`.

```
char char1 = 'A', char2 = 'Z';
cout << "BEFORE swapping chars: " << endl;
cout << "char1: " << char1 << "\tchar2: " << char2 << endl;
swap (char1, char2);
cout << "AFTER swapping chars: " << endl;
cout << "char1: " << char1 << "\tchar2: " << char2 << endl;
```

To:

- Output values and call the `swap()` function to swap the values of `char1` and `char2`.

As the function `swap()` takes two parameters, the order of the arguments is vital. The first argument will be matched to the first parameter, the second argument to the second parameter and so on. I.e. `char1` → `ch1` and `char2` → `ch2`.

```
Compile and run the program.
=============================================================
BEFORE swapping chars:
char1: A        char2: Z
```

```
AFTER swapping chars:
char1: A        char2: Z
```

As you can see, the values of the two variables, `char1` and `char2`, remain unaltered. This is because the parameters are *passed by value* and any alteration to the variables in the function `swap()` do not affect the variables in `main`.

Evidently, `char1` and `char2` are to have their values swapped when the function returns. To ensure that this does occur the parameters should be defined as ***pass by reference***. That is, pass to the parameter variable a reference to the argument variable instead its value. I.e. the address of the argument is passed. Thus, the parameter and the argument are in effect the *same* variable. For example:

Pass by Reference:

char1 Address:0xff0000	ch1 Address:0xff0000
char2 Address:0xff0001	ch2 Address:0xff0001

This is achieved by using the address (`&`) of operator in the prototype *and* definition.

Modify the prototype to:

```
void swap (char &ch1, char &ch2);
```

To:

- Declare the function `swap()` with two reference parameters.

The address (`&`) of operator is inserted *after* the type and *before* the name of each reference parameter.

Modify the definition header to:

```
void swap (char &ch1, char &ch2)
```

To:

- Define the function `swap()` with two reference parameters.

```
Compile and run the program.
BEFORE swap:
char1: A        char2: Z
AFTER swap:
char1: Z        char2: A
```

Now, the values of `char1` and `char2` have been swapped as expected.

Add the following code to main:

```
cout << "BEFORE swapping chars: " << endl;
```

```
cout << "char1: " << 'D' << "\tchar2: " << char2 << endl;
swap ('D', char2);
cout << "AFTER swapping chars: " << endl;
cout << "char1: " << 'D' << "\tchar2: " << char2 << endl;
```

```
Compile the program.
=====================================================
Note:
An error similar to the following will be produced:
functionPassByRef.cpp: In function 'int main()':
functionPassByRef.cpp:18: error: no matching function for call
to 'swap(char, char&)'
functionPassByRef.cpp:5:    note:    candidates    are:    void
swap(char&, char&)
/packages/gcc/4.0.2/bin/../lib/gcc/sparc-sun-
solaris2.9/4.0.2/../../../../include/c++/4.0.2/bits/stl_algoba
se.h:92: note:                          void std::swap(_Tp&, _Tp&)
[with _Tp = char]
```

The error produced during compilation is caused by passing an argument to a reference parameter which does not have an address. I.e. The argument 'D' does not reside in memory. Only variables and named constants are assigned memory. Hence, for reference parameters, the argument must either be a named constant or variable.

```
Remove the code causing the error.
```

Parameters - Default

Parameters can be defined with default values; that is, you can specify a value that will be used if the call does not provide an argument for a particular parameter.

Create a file named "functionDefaultParam.cpp" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the main function.

Add the following function prototype.

```
float calcPay (float rate, float hours = 40, float bonus = 0);
```

To:

- Declare the function `calcPay()` which includes two default parameters.

The parameters with default values must be right-most in the parameter list.

Add the following function definition.

```
float calcPay (float rate, float hours, float bonus)
{
   return hours * rate + bonus;
}
```

To:

- Define the function `calcPay()`.

Add the following code to main:

```
cout << "Pay is $" << calcPay (30.00, 40, 0) << endl;
```

To:

- Call the function `calcPay()` with the arguments `30.0` for `rate`, `40` for `hours` and `0` for `bonus`.

```
Compile and run the program.
=============================================================
Pay is $1200
```

Add the following code to main:

```
cout << "Pay is $" << calcPay (30.00, 40) << endl;
```

To:

- Call the function `calcPay()` with the arguments `30.0` for `rate` and `40` for `hours`. The default value for `bonus` will be used.

```
Compile and run the program.
=============================================================
Pay is $1200
Pay is $1200
```

Add the following code to main:

```
cout << "Pay is $" << calcPay (30.00) << endl;
```

To:

- Call the function with the arguments `30.0` for `rate`. The defaults for `hours` and `bonus` will be used.

```
Compile and run the program.
```

```
Pay is $1200
Pay is $1200
Pay is $1200
```

It is important to note that the omitted arguments apply to the right-most parameter first. For example, it is not possible to omit the `hours` argument only. If the default for `hours` was to be used, then the default for `bonus` must also be used.

```
Close the file.
```

Overload

It is possible to have multiple functions defined with the same name. As long as the parameter list differs in type, the compiler can determine which function to invoke.

```
Save the current file as "functionOverload.cpp".
```

Add the following function prototype:

```
void swap (float &f1, float &f2);
```

To:

- Declare an overloaded function to `swap()` which takes float reference parameters.

Add the following function definition:

```
void swap (float &f1, float &f2)
{
   float tmp = f1;
   f1 = f2;
   f2 = tmp;
}
```

To:

- Define an overloaded function to `swap()`.

Add the following code to `main`:

```
float float1 = 1.5, float2 = 5.1;

cout << "BEFORE swapping floats: " << endl;
cout << "float1: " << float1 << "\tfloat2: " << float2
     << endl;
swap (float1, float2);
cout << "AFTER swapping floats: " << endl;

cout << "float1: " << float1 << "\tfloat2: " << float2
```

```
        << endl;
```

To:

- Swap two float variables and print the results.

When the function call to `swap()` is encountered with `char` arguments, the matching function is invoked. Likewise, the call with `float` arguments invokes the function with `float` parameters.

Compile and run the program.

```
BEFORE swapping chars:
char1 : A       char2  : Z
AFTER swapping chars:
char1 : Z       char2  : A
BEFORE swapping floats:
float1: 1.5     float2: 5.1
AFTER swapping floats:
float1: 5.1     float2: 1.5
```

In the previous example, type conversions were performed where necessary by the compiler. This was possible as the parameters were passed by value. It is important to note, however, that type conversions are not possible with reference parameters.

Close the file.

Programming Exercise:
Write a program that extends the previous exercise and make it modular.

1. Create a C++ program named "`practiceExerciseFunction.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Copy the code from "`practiceExerciseDoWhile.cpp`".
4. Declare the following function prototypes.

```
int largest (int, int);
int smallest (int, int);
bool validInput (int &);
```

5. Define the functions accordingly:

`largest`: Compares the two integer parameters and returns the value which is the largest.

`smallest`: Compares the two integer parameters and returns the value that is the smallest.

`getInput`: Receives user input into the parameter variable and

returns true if the input is valid, or false if it is not. The parameter is pass by reference thus, the value will be returned to the calling function.

6. Cut the code in the do-while loop which determines the *largest* value and paste it into the function `largest`. Place a function call to `largest()` in its place and assign the value returned by the function to the appropriate variable.
7. Cut the code in the do-while loop which determines the *smallest* value and paste it into the function `smallest`. Place a function call to `smallest()` in its place and assign the value returned by the function to the appropriate variable.
8. Cut the code which receives user input into the current value variable and place it inside the function `getInput`.
   * Test if the value is valid and return true if it is otherwise return false. The variable will be returned to the calling function through the `int&` parameter.
   * Place a function call to `getInput` in its place and test the value returned by `getInput` for `true/false`. If true, the input was valid, otherwise it was not.
   * If it is not valid, ignore it in the largest, smallest, etc calculations but rememeber to increment the count of invalid input.
9. The rest of the code remains the same.
10. Output the results.
11. Compile and run the program to ensure that the code is logically correct.

Array Data Structure

Single Dimension

An array is a data structure which stores multiple data of the same type in contiguous memory locations. That is, one after the other. Hence, the addresses in memory are sequential. The advantage of using an array to store *N* items rather than *N* individual variables is that only one identifier is required. This is particularly beneficial when passing data to functions. The constructs of array usage makes it possible to access each of the components to the array inside a loop.

Declaration

Create a file named "`arrayBasic.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

An array can be declared as any of the built-in or user-defined types and the size of the array must be stated during the declaration. Each individual data item in the array is called an element. Add the following code to `main`:

```
int iArray[5];
```

To:

- Declare an `int` array with 5 elements. That is, its size is 5.

`iArray` can hold up to 5 integer values. The size of the array must be a constant value i.e. `5` or `SIZE` (where `SIZE` is a declared constant such as `const int SIZE = 5;`) Hence, you *cannot* use a variable to declare the size.

Add the following code to `main`:

```
for (int i = 0; i < 5; i++)
   cout << "Address of iArray[" << i << "] = " << &iArray[i]
      << endl;
```

To:

- Print the *address* of each of the array elements.

To access each of the elements in the array, the array subscript (`[]`) operator is used. The *index* number of the element is placed between the `[]`. The elements in the array are indexed from 0 (first) to size-1 (last).

`iArray:`

Index	0	1	2	3	4
Address:	0xFF00	0xFF04	0xFF08	0xFF0C	0xFF10
Identifer	iArray[0]	iArray[1]	iArray[2]	iArray[3]	iArray[4]

```
Compile and run the program.
=============================================================
Address of iArray[0] = 0xff00
Address of iArray[1] = 0xff04
Address of iArray[2] = 0xff08
Address of iArray[3] = 0xff0c
Address of iArray[4] = 0xff10
=============================================================
Note:
The output on your machine will differ from this.
```

Notice that the addresses of the elements are sequential in 4 byte increments, which is the number of bytes per `int`.

Add the following code to `main`:

```
for (int i = 0; i < 5; i++)
   cout << "Value of iArray[" << i << "] = " << iArray[i]
      << endl;
```

To:

- Print the *value* of the 10 array elements.

Comment out the code to print the addresses.

```
/*for (int i = 0; i < 5; i++)
   cout << "Address of iArray[" << i << "] = " << &iArray[i] <<
endl;*/
```

```
Compile and run the program.
=============================================================
Value of iArray[0] = 0
```

```
Value of iArray[1] = 0
Value of iArray[2] = 0
Value of iArray[3] = -4197072
Value of iArray[4] = 67932
```

Notice that the values in the array are *odd*. This is because the current content of the respective memory locations is leftover from some previous application.

It is possible to assign initial value(s) to an array during the declaration statement. Modify the code `int iArray[5];` to:

```
int iArray[5] = {0, 1, 2, 3};
```

To:

- Declare an `int` array with 5 elements and assign the values `{0, 1, 2, 3}` to the first 4 elements.

```
Compile and run the program.
=============================================================
Value of iArray[0] = 0
Value of iArray[1] = 1
Value of iArray[2] = 2
Value of iArray[3] = 3
Value of iArray[4] = 0
```

When initial values are given to an array the values are assigned sequentially to the elements. If there are less than *size* values, the remaining elements will be initialized to the value `0`.

Modify the code to print the array contents to :

```
for (int i = 0; i < 1000; i++)
   cout << "Value of iArray[" << i << "] = " << iArray[i] <<
endl;
```

To:

- Print the *value* of 1000 array elements.

```
Compile and run the program.
=============================================================
Value of iArray[0] = 0
Value of iArray[1] = 1
Value of iArray[2] = 2
Value of iArray[3] = 3
Value of iArray[4] = 0
Value of iArray[5] = 0
Value of iArray[6] = 0
Value of iArray[7] = 0
```

```
Value of iArray[8] = 0
Value of iArray[9] = 0
...
...
Value of iArray[700] = 5461326
Value of iArray[701] = 1462523244
Value of iArray[702] = 1953653037
Value of iArray[703] = 872440178
Value of iArray[704] = 1919050099
Value of iArray[705] = 0
Value of iArray[706] = 0
Segmentation Fault
```

```
Note:
The output on your machine will differ.
The program has crashed.
```

All systems are different but if the value for the index is great enough, the program should *crash* due to a memory violation. That is, accessing memory not allocated to your program. The array has only been allocated 5 integer memory locations and the code is attempting to access well beyond that size.

Hence, be very careful about accessing memory not allocated to your array. Modify the program to:

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

const int SIZE = 5;

int main ()
{
   int iArray[SIZE] = {0,1,2,3};

   for (int i = 0; i < SIZE; i++)
      cout << "Value of iArray[" << i << "] = " << iArray[i]
         << endl;

   return 0;
}
```

To:

- Declare a constant identifier to use in the declaration statement for the array's size and also in the `for` loop, iterating through each element.

By using a constant to declare the size of the array and as the upper bound element number, the code becomes less error prone.

```
Compile and run the program.
```

```
Value of iArray[0] = 0
Value of iArray[1] = 1
```

```
Value of iArray[2] = 2
Value of iArray[3] = 3
Value of iArray[4] = 0
```

As Function Parameter

A parameter to a function can be an array. By default, arrays are passed to functions by reference and there is no way to alter this.

Add the following function prototype:

```
void fillArray (int []);
```

To:

Declare a function named `fillArray()` that takes an array of type `int` as its parameter

When declaring an array as a parameter, the `[]` **must** be included after the type to indicate that it is an array. Also, the size of the array is not required.

Add the following function prototype:

```
void fillArray (int intArray[])
{
   for (int i = 0; i < SIZE; i++)
   {
      cout << "Enter value " << i+1 << ": ";
      cin >> intArray[i];
   }
}
```

To:

- Define the function named `fillArray()`.

Notice that the address of (`&`) operator is not included in the parameter list for arrays. Remember, arrays are passed by reference by default so there is no need to explicitly state this. Also, the `[]` is placed after the identifier.

Add the following function call to `main` *before* the code to print the array.

```
fillArray (iArray);
```

To:

- Call the function `fillArray()` and pass the argument, `iArray`.

The argument to the function call is the same as for any variable - identifier only.

```
Compile and run the program.
```

```
Enter value 1: [100]
```

```
Enter value 2: [45]
Enter value 3: [32]
Enter value 4: [159]
Enter value 5: [357]
Value of iArray[0] = 100
Value of iArray[1] = 45
Value of iArray[2] = 32
Value of iArray[3] = 159
Value of iArray[4] = 357
```

Individual array elements can be passed to an array. When doing so, the default parameter passing reverts to *pass by value*.

Add the following function prototype:

```
void printElement (int, int);
```

To:

- Declare a function named `printElement()` that takes two `int` parameters.

Add the following function definition:

```
void printElement (int i, int element)
{
   cout << "Value of element[" << i << "] = " << element
      << endl;
}
```

To:

- Define the function named `printElement()` and print the value of the element passed to it.

Now, instead of passing the array, an individual element is passed.

Modify main to:

```
int main ()
{
   int iArray[SIZE] =  {0,1,2,3,4,5};

   fillArray (iArray);

   for (int i = 0; i < SIZE; i++)
      printElement (i, iArray[i]);

   return 0;
}
```

To:

- Replace the code to print the entire array with a function call to `printElement()` which will print one element of the array only. This will be called `SIZE` times.

While `iArray` is an array of 10 integers, `iArray[i]` is one individual integer.

When passing `iArray` to a function, a reference to the array is passed. I.e. The entire array is accessible within the function as is the case with the function `fillArray()`.

When passing `iArray[i]` to a function, the value of `iArray[i]` is assigned to the parameter variable, `element` when it is *pass by value*. In *pass by reference*, `iArray[i]` is accessible by the function but not the other elements.

Compile and run the program.

```
Enter value 1: [10]
Enter value 2: [9]
Enter value 3: [8]
Enter value 4: [7]
Enter value 5: [6]
Value of element[0] = 10
Value of element[1] = 9
Value of element[2] = 8
Value of element[3] = 7
Value of element[4] = 6
```

Close the file.

Multi-dimension

The arrays explored so far are referred to as single dimension arrays. That is, they have one dimension only. However, an array can have more than one dimension. This is referred to as a multi-dimensional array. It is in effect, an array of arrays.

Generally speaking, there is no restriction to the number of dimensions, although in practice one can only code what can be effectively conceptualized.

They are very similar to standard arrays with the exception that they have multiple sets of subscript operators `[]`. That is, one for each dimension.

Declaration

Create a file named "`arrayMultiDimension.cpp`" and add the following code:

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

int main ()
{
```

```
    return 0;
}
```

To:

- Define the `main` function.

Add the following code to `main`:

```
const int N_ROWS = 3, N_COLS = 2;
int iArray2D[N_ROWS][N_COLS] = {{11,12}, {22, 22}};
```

To:

- Declare a two dimensional `int` array with 3x2 elements.

A multi-dimension array can have initial values assigned during its declaration. Like in single dimensional arrays, all unspecified values will be assigned the value `0`.

A two dimensional array can be thought of as a matrix of rows and columns. The first dimension is the row component and the second the column.

iArray2D

	COL		
	INDEX	**0**	**1**
ROW	0	11	12
	1	21	22
	2	0	0

Add the following code to `main`:

```
cout << "  col\t0" << "\t1" << endl;
cout << "row |==========\n";
for (int row = 0; row < N_ROWS; row++)
{
   cout << ' '<< row <<"  |\t";
   for (int col = 0; col < N_COLS; col++)
      cout << iArray2D[row][col] << "\t";
   cout << endl;
}
```

To:

- Print the contents of `iArray2D` in a tabular format.

The code iterates through the array accessing one row at a time. For each row indexed, each column will also be indexed. Hence the order of the array indexing is:

```
          Row Col
    iArray[0] [0]
    iArray[0] [1]
    iArray[1] [0]
    iArray[1] [1]
    iArray[2] [0]
```

```
iArray[2] [1]
```

```
Compile and run the program.
```

```
  col    0        1
row |==============
 0  |    11       12
 1  |    21       22
 2  |    0        0
```

As Function Parameter

Multi-dimensional arrays can be passed to functions in a similar manner as that of single dimension arrays.

Add the following function prototype:

```
void fillArray (int [][], int, int);
```

To:

- Declare a function `fillArray()` which takes one two dimensional `int` array and two `int` parameters.

Add the following function definition:

```
void fillArray (int array[][], int nRows, int nCols)
{
   for (int r = 0; r < nRows; r++)
   {
      for (int c = 0; c < nCols; c++)
      {
         cout << "Enter value for element at row(" << r
              << ") col("<<c<< "): " ;
         cin >> array[r][c];
      }

   }
}
```

To:

- Define the function `fillArray()`.

In this definition, the dimensions of the array are being passed to the function as the function has no way of knowing this information unless it is explicitly given. I.e. the programmer must always explicitly state where the array ends. This applies to single dimension arrays also.

```
Compile and run the program.
```

```
arrayMultiDimension.cpp:6: error: multidimensional array must
have bounds for all dimensions except the first
```

Note:
The program should produce the error message above.

As it explicitly states, for multi-dimensional arrays, the size of the dimensions, except the first, **must** be given.

Modify the function prototype:

```
void fillArray (int [][N_COLS], int, int);
```

And also the header of the definition:

```
void fillArray (int array[][N_COLS], int nRows, int nCols)
```

Add the following code to `main` *before* the code to print the array.

```
        fillArray (iArray2D, N_ROWS, N_COLS);
```

Compile and run the program.

```
Enter value for element at row(0) col(0): [111]
Enter value for element at row(0) col(1): [222]
Enter value for element at row(1) col(0): [333]
Enter value for element at row(1) col(1): [444]
Enter value for element at row(2) col(0): [555]
Enter value for element at row(2) col(1): [666]
  col   0       1
row |============
 0  |   111     222
 1  |   333     444
 2  |   555     666
```

Close the file.

Programming Exercise:
Write a program that searches an array of integers for a value and if found, returns its index number, otherwise returns -1.

1. Create a C++ program named "`practiceExerciseArray.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Add the following function prototype:
   `void findArray (int[], int, int);`

4. Add the following partial function definition:

```
void findArray (int array[], int size, int val)
{
}
```

The size parameter indicates the number of elements in the array to sort. An algorithm for sort the array is.
The val parameter is the value to search for in the array.

The following algorithm should be implemented

```
FOR index = 0 to size-1
        IF current element is the value
                RETURN index value
        END IF
END FOR
RETURN -1
```

5. Declare an int array of size 10 and receive user input to populate its data. I.e.
6. Declare an int variable to hold the value to be searched for in the array. This value will be entered by user input.
7. Call the function `findArray`, passing the array and its size along with the value to search for.
8. Assign the returned value to a variable which represents the index number of the array the value was found in. If the value returned is -1, the value wasn't found, otherwise it was.
9. Output an appropriate message to indicate whether it was found or not.
10. Compile and run the program to ensure that the code is logically correct.

C-String

Arrays can be declared as any defined type e.g. `int`, `float`, `char`. However, when the type is `char` the array can *also* be treated as a *string* of characters rather than multiple single characters.

For example, it is not possible to output an entire `int` array in one instance. Instead, each element must be output separately. However, when the array is a `char` array the compiler can output the array as a whole in one instance.

Create a file named "`cString.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

The only difference between an array of characters and a c-string is that the last character in the c-string must be the null (`\0`) character. This indicates to the compiler the end of the c-string.

Declaration

Add the following code to `main`:

```
char cstr1[50];
char cstr2[50] = "";
char cstr3[50] = "this is cstr3";
```

To:

- Declare three c-string variables and assign initial values to two

When assigning initial values during the declaration statement, the null character `\0` is automatically appended to end of the c-string.

Add the following code to `main`:

```
char cstr4[] = "this is cstr4";
```

To:

- Declare a c-string variable without an explicit size with "`this is cstr4`" as the initial value.

When an array (`char` or other) is declared and the size is not explicitly stated, the size will be determined by the number of values assigned. Thus, the size of `cstr4` will be the `sizeof ("this is cstr4")` i.e. 13 + 1 for the null character.

Output

The stream insertion (<<) operator can be used to output c-strings. It works outputting all characters from the beginning of the array (c-string) until the `\0` is encountered.

Add the following code to `main`:

```
cout << "cstr4: " << cstr4 << endl;
cout << "cstr3: " << cstr3 << endl;
cout << "cstr2: " << cstr2 << endl;
cout << "cstr1: " << cstr1 << endl;
```

To:

- Print the four c-string variables.

When outputting a c-string only the name of the variable is used. i.e. the `[]` are omitted.

```
Compile and run the program.
=============================================================
cstr4: this is cstr4
cstr3: this is cstr3
cstr2:
cstr1: ¶╕ ├wΦ?=
=============================================================
Note:
The output for cstr1 will differ on your machine.
```

Notice that the last two c-string outputs:

- As `cstr2` was initialised to "" no characters are output as the first element will contain the `\0`.
- The output for `cstr1` is *odd* and unpredictable. The issue with this output is that `cstr1` is un-initialized and as such, whatever values were in the memory location previously are being interpreted as characters and outputted. It seems that a `\0` character was by chance encountered in element 8 of `cstr1`.

Input

The stream extraction (>>) operator can be used to input into a c-string variable with similar rules as apply for other type input. I.e. The input is de-limited by white-space.

Add the following code to `main`:

```
cout << "Enter a value for cstr1: ";
cin >> cstr1;
```

```
cout << "Enter a value for cstr2: ";
cin >> cstr2;
cout << "cstr1: " << cstr1 << endl;
cout << "cstr2: " << cstr2 << endl;
```

To:

- Receive input into the two c-string variables.

Once the characters have been extracted and stored in the c-string variables, the null character will be appended to the end of each c-string.

```
Compile and run the program.
=============================================================
cstr4: this is cstr4
cstr3: this is cstr3
cstr2:
cstr1: ¶╕ ├w♦?=
Enter a value for cstr1: [this is cstr1]
Enter a value for cstr2:
cstr1: this
cstr2: is
```

As input using (>>) is de-limited by white-space, the string "`this`" is extracted and assigned to `cstr1` and "`is`" is assigned to `cstr2`. "`cstr1`" remains in the input stream and will be the next data extracted.

If white-space is to be included in the c-string, the `getline()` function can be used.

Replace the code `cin >> cstr1;` and `cin >> cstr2;` with the following:

```
cin.getline (cstr1, sizeof (cstr1), '\n');
cin.getline (cstr2, sizeof (cstr2), '\n');
```

The `getline()` function extracts characters, including white-space , from the input stream and stores them in the given c-string variable. Extraction will stop when either `n-1 (sizeof(cstr1))` characters have been extracted or the de-limiter (`\n`) is encountered.

The de-limiter will be extracted from the input stream and discarded and the null character will be appended to the end of the c-string.

In this case, extraction will continue until either 49 characters have been extracted or the `\n` character is encountered. The de-limiter can be any character although it is very often the `\n` character.

```
Compile and run the program.
=============================================================
cstr4: this is cstr4
cstr3: this is cstr3
cstr2:
cstr1: ¶╕ ├w♦?=
Enter a value for cstr1: [this is cstr1]
```

```
Enter a value for cstr2: [this is cstr2]

cstr1: this is cstr1
cstr2: this is cstr2
```

Manipulation

The assignment operator (=) can only be used with c-strings during an initialization statement. In order to alter the value of a c-string the c-string functions can be used.

Add the following `include` statement *before* `main`:

```
#include <cstring>
```

To:

- Use the string manipulation functions.

Add the following code to `main`:

```
strcpy (cstr1, "cstr1");
cout << "\nstrcpy (cstr1, \"cstr1\");" << endl;
```

To:

- Copy the string "`cstr1`" into `cstr1`.

The size of `cstr1` remains 50 but the value is now "`cstr1\0`". The null character will be appended to the end of the c-string.

The `sizeof()` function returns the size of the c-string. That is, how many elements are in the array. This does *not* refer to the number of valid characters in the c-string.

There are many c-string functions defined to manipulate c-strings. The following are the most commonly used c-string functions.

`strlen(cstr)`	Returns the length of `cstr`. i.e. how many characters are before the `\0`
`strcat (cstr1, cstr2)`	Appends `cstr2` to `cstr1`.
`strcpy (cstr1, cstr2)`	Copies `cstr2` to `cstr1`.
`strcmp (cstr1, cstr2)`	Compare `cstr1` and `cstr2` and return the difference. If they are the same, `0` is returned.

Add the following code to `main`:

```
cout << "The size of cstr1 is  : " << sizeof (cstr1) << endl;
cout << "The length of cstr1 is: " << strlen (cstr1) << endl;
```

```
cout << "cstr1: " << cstr1 << endl;
```

To:

- Output the size and length of `cstr1`.

```
Compile and run the program.
=============================================================
cstr4: this is cstr4
cstr3: this is cstr3
cstr2:
cstr1: ╫├┬w↑ "
Enter a value for cstr1: [this is cstr1]
Enter a value for cstr2: [this is cstr2]

cstr1: this is cstr1
cstr2: this is cstr2

strcpy (cstr1, "cstr1");
The size of cstr1 is  : 50
The length of cstr1 is: 5
cstr1: cstr1
```

C-strings can be concatenated. That is, a c-string can have another c-string added to its end .

Add the following code to `main`:

```
strcat(cstr1,"-make sure it's not longer than 50 characters");
cout << "\nstrcat(cstr1,\"-make sure it's not longer than 50
characters\");" << endl;
cout << "The length of cstr1 is: " << strlen (cstr1) << endl;
cout << "cstr1: " << cstr1 << endl;
```

To:

- Add the end of `cstr1` the string `"-make sure it's not longer than 50-1 chars"` and output it.

```
Compile and run the program.
=============================================================
cstr4: this is cstr4
cstr3: this is cstr3
cstr2:
cstr1: ╫├┬w↑ "
Enter a value for cstr1: [this is cstr1]
Enter a value for cstr2: [this is cstr2]

cstr1: this is
cstr2: this is
```

```
strcpy (cstr1, "cstr1");
The size of cstr1 is  : 50
The length of cstr1 is: 5

cstr1: cstr1

strcat(cstr1,"-make sure it's not longer than 50-1 chars");
The length of cstr1 is: 47
cstr1: cstr1-make sure it's not longer than 50-1 chars
```

```
close the file.
```

Programming Exercise:

Write a program that receives a line of input from the user then counts how many vowels, consonants and punctuation are in it

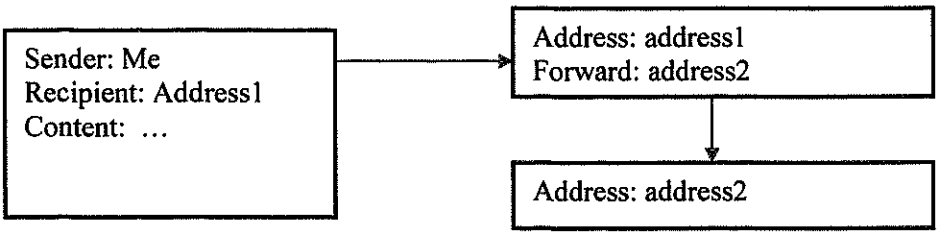
1. Create a C++ program named "`practiceExerciseCString.cpp`".
2. Add the `main` function and the appropriate include file(s).
3. Add the following function prototype:
   ```
   void statistic (char [], int &v, int &c, int &p);
   ```
4. Add the function definition.
   The code in the function parses the line of input, stopping when it reaches the end of the string and keeps count of how many vowels (a e i o u), consonants (other characters) and how many punctuation characters (e.g . ? , ' " etc) it encountered. The case of the characters is ignored. i.e. 'A' and 'a' are the same. The count values are returned via the reference parameters.
5. Declare a c-string variable large enough to accept a line of input.
6. Receive from user input the line of input.
7. Call the function `statistic()` passing the c-string, and the three variables to store the counted values in.
8. Output the string along with the values of each count.
9. Compile and run the program to ensure that the code is logically correct.

Pointer Data Type and Pointer Variables

A pointer is a special *type* whose value is a *reference* to another value stored elsewhere in the computer memory. That is, its value is the *address* of another memory location.

A pointer can be a reference to any built-in or user-defined type.

An analogy to pointers could that of mail forwarding. Consider two email addresses `address1` and `address2` where `address1` has its mail forwarded to `address2`. When an email addressed to `address1` arrives at `address1`, it is forwarded to `address2` without `address1` accessing its content. The diagram below illustrates this point.



The value in a pointer can be thought of as a forwarding address.

Declaration, De-Referencing and Assignment

This section illustrates how to declare, assign values to and de-reference pointers.

Create a C++ program named "`pointersIntro.cpp`" and add the following code:

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

int main ()
{

   return 0;
}
```

To declare a pointer variable an asterisk (*) is inserted *after* the type and *before* the variable name.

Add the following code to `main`:

```
int * iPtr;
```

To:

- Declare a pointer variable named `iPtr`

`iPtr` can reference (*point to*) memory locations that hold data of type `int`.

There are three values associated with a pointer.

- Its address in memory.
- Its content, which is an address in memory of the data that it references.
- The value contained in the memory location it references.

To access the address of a pointer (or any data item), the address (`&`) of operator is added *before* the name.

Add the following code to `main`:

```
cout << "&iPtr: " << &iPtr << " :address of iPtr\n";
```

To:

- Print the address in memory of `iPtr`. That is, its *own* address.

To access the content of a pointer (the address of the memory it references), the name of the pointer alone is used.

Add the following code to `main`:

```
cout << "iPtr:  " << iPtr << " :content of iPtr\n";
```

To:

- Print the value stored in `iPtr`. That is, the address of the memory location it references.

To access the content of the memory location referenced by a pointer, the de-reference (*) operator is used.

Add the following code to `main`:

```
cout << "*iPtr: " << *iPtr
   << " :value referenced by iPtr\n";
```

To:

- Print the value in the memory location referenced by `iPtr`.*Not e: Accessing values referenced by pointers is called de-referencing.*

```
Compile and run the program.
=============================================================
&iPtr: 0xffbcff11 :address of iPtr
iPtr:  0xff001111 :content of iPtr
*iPtr: -123456789 :value referenced by iPtr
    OR
&iPtr: 0xffbcff11 :address of iPtr
iPtr:  0xff001111 :content of iPtr
Segmentation fault (core dump)
    OR
Something similar (i.e. the program crashes)
=============================================================
Note:
The values produced on your own machine will differ.
```

In the output above, the first two statements produce values beginning with `0x`, which is hexadecimal notation and represents memory location addresses. Both `&iPtr` and `iPtr` produce output which represent addresses. The content of `iPtr` is interpreted as an address because `iPtr` is a pointer variable and pointer variables can *only* hold addresses.

For the last output statement, either the output will be some value which is *unusual* or the program will *crash* during execution due to a memory violation. That is, access to memory which has not been allocated to the program. The current value in `iPtr`, which is leftover from some previous application, represents memory which has not been allocated to the program and as such may cause serious problems during execution if accessed.

Thus, whenever a pointer variable is not set to a valid memory location the *null* value, 0, should be used. A constant named `NULL`, whose value is 0, has been defined for this purpose.

Add the following statement directly after the `#include <iostream>` directive,

```
#include <cstdlib>
```

To:

- Enable use of `NULL`.

Add the following code to `main`, *before* the three `cout` statements,

```
iPtr = NULL;
```

To:

- Set the value of `iPtr` to *null*.

Always set pointer variables to null whenever they are not referencing valid memory. Now, before accessing the memory referenced by `iPtr`, a test can be performed to check if its value is *null*. Modify the program by replacing the following statement,

```
cout << "*iPtr: " << *iPtr
     << " :value referenced by iPtr\n";
```

with ,

```
if (iPtr != NULL)
   cout << "*iPtr: " << *iPtr
      << " :value referenced by iPtr\n";
```

To:

- Check whether `iPtr` is set to *null* and print the value its contents if not.

`iPtr` is now protected from accessing unallocated memory and the program should run successfully.

```
Compile and run the program.
```

```
&iPtr: 0xffbcff11 : address of iPtr
iPtr:  0xff001111 : content of iPtr
```

```
Note:
The values produced on your own machine will differ from this.
```

```
Close the file.
```

Create a C++ program named "`pointersAssign.cpp`" and add the following code:

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

int main ()
{

   return 0;
}
```

However, it is intended that legitimate memory be assigned to pointer variables and this is done with the assignment (=) operator.

Add the following statements to `main`,

```
int iVar = 0;
iPtr = iVar;
```

To:

- Declare an integer variable named `iVar` and set its value to `0`.

Assign the value of `iVar` to `iPtr`.

Compile the program.

Note:
A compile error should occur.

The program should not compile successfully as it is invalid to assign values, except 0 (*null*), directly to pointer variables. Only the addresses of data in memory can be assigned. This is enabled by the address of (`&`) operator. Modify the statement,

```
iPtr = iVar;
```

to the following statement,

```
iPtr = &iVar;
```

To:

- Assign the address of `iVar` to `iPtr`.

`iPtr` now references (points to) `iVar`.

Compile the program.

The program should now compile.

Add the following code to `main`:

```
cout << "iPtr  : " << iPtr << endl;
cout << "&iVar : " << &iVar << endl;
```

To:

- Output the value of `iPtr` and the address of `iVar`.

Compile the program.

```
    0xffcde110
    0xffcde110
```

```
Note:
The values produced on your own machine will differ from this.
```

Note that the values output are equivalent, as you would expect, as the value in `iPtr` is the address of `iVar`.

As discussed previously, in order to access the data associated with a pointer, the de-reference `(*)` operator is used.

Add the following code to `main`,

```
*iPtr = 30;
```

To:

- Access the data stored in the memory location referenced by `iPtr` (`iVar`) and set its value to `30`.

Add the following code to `main`:

```
cout << "*iPtr : " << *iPtr << endl;
cout << "iVar  : " << iVar << endl;
```

To:

- Output the value de-referenced by `iPtr` and the value of `iVar`.

```
Compile and run the program.
```

```
    *iPtr: 30
    iVar : 30
```

Note the output. The values of `*iPtr` and `iVar` are equivalent.

By now, it should be very clear that there are two ways to access the value stored in `iVar`; either through `iVar` itself or by de-referencing `iPtr`. At this point it seems unnecessary to use a pointer to access a variable when the variable can be accessed directly. Actually, it is! However, later in this topic we will introduce dynamic memory and this is where pointers become useful and powerful.

```
Close the file.
```

Create a C++ program named "`pointersMultiple.cpp`" and add the following code:

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

int main ()
{

 return 0;
}
```

When declaring more than one pointer in a declaration statement, the (*) must be placed before each pointer variable name.

Add the following code to `main`:

```
int *iPtr1 = NULL, *iPtr2 = NULL, iVar1, iVar2;
iPtr1 = &iVar1;
iPtr2 = &iVar2;

*iPtr1 = 10;
*iPtr2 = 20;

cout << "*iPtr1: " << *iPtr1 << endl;
cout << "*iPtr2: " << *iPtr2 << endl;
```

To:

- Declare two integer pointer variables named `iPtr1`and `iPtr2` and two integer variables named `iVar1` and `iVar2`.
- Assign the addresses of `iVar1` and `iVar2` to `iPtr1` and `iPtr2` respectively.
- Set the value de-referenced by `iPtr1` to `10`.
- Set the value de-referenced by `iPtr2` to `20`.
- Output the value de-referenced by `iPtr1`.
- Output the value de-referenced by `iPtr2`.

```
Compile and run the program.
```

```
*iPtr1: 10
*iPtr2: 20
```

A pointer variable can be assigned the value of another pointer.

Add the following code to `main`:

```
iPtr1 = iPtr2;
cout << "*iPtr1: " << *iPtr1 << endl;
cout << "*iPtr2: " << *iPtr2 << endl;
```

To:

- Assign the value of `iPtr2` to `iPtr1` and output there de-referenced values.

Remember, a pointer variable can only be assigned the address of a data item or the *null* value. As `iPtr2` contains an address as its value, the statement:

```
iPtr1 = iPtr2;
```

Is legal and ensues that `iPtr1` and `iPtr2` reference the same memory location. Thus, changes made to `*iPtr1` effect `*iPtr2` and vice versa.

Compile and run the program.

```
*iPtr1: 10
*iPtr2: 20
*iPtr1: 20
*iPtr2: 20
```

Note that in the second set of output statements, `*iPtr1` and `*iPtr2` now both produce the value `20`.

Add the following statements to `main`:

```
*iPtr1 = 100;
cout << "*iPtr1: " << *iPtr1 << endl;
cout << "*iPtr2: " << *iPtr2 << endl;
```

To:

- Modify the value de-referenced by `iPtr1` to `100` and output their respective de-referenced values.

Compile and run the program.

```
*iPtr1: 10
*iPtr2: 20
*iPtr1: 20
*iPtr2: 20
*iPtr1: 100
*iPtr2: 100
```

Close the file.

Summary:

- `&iPtr` Access the address of `iPtr` (`iPtr's` location in memory).
- `iPtr` Access the content of `iPtr` (address of another data instance in memory).
- `*iPtr` Access the content of the memory location referenced by `iPtr`.

Programming Exercise:

Create a program that swaps two integer variables using pointers.

1. Create a C++ program named "practiceExercisePointer.cpp".
2. Add the `main` function and the appropriate include files.
3. Declare two integer pointer variables named `iPtr1` and `iPtr2`.
4. Declare two integer variables named `iVar1` and `iVar2` and initialize them to the values, `10` and `20`.
5. Assign `iVar` to `iPtr1` and `iVar2` to `iPtr2`.
6. Output the values de-referenced by `iPtr1` and `iPtr2`.
7. Add the necessary code to swap the values de-referenced by `iPtr1` and `iPtr2` by using a third variable to hold the data during the swap.
8. Output the values de-referenced by `iPtr1` and `iPtr2`.
9. Compile and run the program.

Note: The code should swap the values in iVar1 and iVar2 indirectly. I.e. use the pointers.

With Relational Operators

Create a program named "`pointersRelational.cpp`" and add the following:

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

int main ()
{

   return 0;
}
```

The relational operators (`< <= > >= == !=`) can be used with pointers, however, be careful to use appropriately.

A common programming error is to compare the pointer reference instead of the value de-referenced by the pointer. Sometimes though, it may be the actual reference that needs to be compared.

Add the following code to `main`:

```
int *iPtr1 = NULL, *iPtr2 = NULL;
int iVar1 = 0, iVar2 = 0;

iPtr1 = &iVar1;
```

```
iPtr2 = &iVar2;
```

To:

- Declare two integer pointer variables named `iPtr1` and `iPtr2`.
- Declare two integer variables named `iVar1` and `iVar2`.
- Assign the address of `iVar1` to `iPtr1`.
- Assign the address of `iVar2` to `iPtr2`.

`iPtr1` references `iVar1` and `iPtr2` references `iVar2`.

Add the following code to `main`:

```
cout << "Enter two integer values:";
cin >> *iPtr1 >> *iPtr2;

cout << "Common error - comparing addresses\n";
if (iPtr1 == iPtr2)
   cout << setw (8) << iPtr1 << " == " << setw (8)
      << iPtr2 << endl;
else
   cout << setw (8) << iPtr1 << " != " << setw (8)
      << iPtr2 << endl;

cout << "\nCorrect comparison - values\n";
if (*iPtr1 == *iPtr2)
   cout << setw (8) << *iPtr1 << " == " << *iPtr2
      << endl;
else
   cout << setw (8) << *iPtr1 << " != " << *iPtr2
      << endl;
```

To:

- Receive user input into the variables de-referenced by `iPtr1` and `iPtr2`.
- Compare the addresses stored in `iPtr1` and `iPtr2` for equality and output a message indicating whether they are equal or not.
- Compare the values de-referenced by `iPtr1` and `Ptr2` for equality and output a message indicating whether they are equal or not.

```
Compile and run the program.
```

```
Enter two integer values: [8 8]
Common error - comparing addresses
0xff0011cd != 0xff0011d1

Correct comparison - values
8 == 8
```

Note:

```
The address values produced on your own machine will differ.
```

Summary:

- To perform relational operations on the values de-referenced by pointers, use the de-reference (*) operator.
- To perform relational operations on the address values in pointers, *omit* the de-reference (*) operator.
- Ensure that the pointers are of the same type.

```
Close the file.
```

Other Operations on Pointer Variables

Create a program named "`pointersOperations.cpp`" and add the following code:

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

int main ()
{

   return 0;
}
```

So far the operations that can be performed on pointers are that of:
Assignment (=)
Relational operations (< <= > >= == !=)

It is possible to perform other limited arithmetic operations on pointers. The most common operations is that of increment (++) and decrement (--).

Add the following code to `main`:

```
char chArray[] = {'a', 'b', 'c', '\0'};
char *chPtr = chArray;
```

To:

- Declare a character array named `chArray` of size 4 containing the values 'a', 'b', 'c', '\0';
- Declare a character pointer variable named `chPtr` and assign `chArray` to it.

When an array is assigned to a pointer, it is the base address of the array that is assigned. That is, the first element in the array, or in this case, `chArray[0]` .

Add the following code to `main`:

```
    cout << *chPtr;
```

To:

- Output the value de-referenced by `chPtr`.

```
Compile and run the program.
============================================================
a
```

But how to use `chPtr` to access the other elements in the array? This can be achieved by using the increment (++) operator.

Add the following code to `main`:

```
    chPtr++;
    cout << *chPtr;
    chPtr++;
    cout << *chPtr;
```

To:

- Increment `chPtr` to the next element in the array.
- Output the value de-referenced by `chPtr`.
- Increment `chPtr` to the next element in the array.
- Output the value de-referenced by `chPtr`.

```
Compile and run the program.
============================================================
abc
```

The same can be applied with the decrement (--) operator.

Add the following code to `main`:

```
    chPtr--;
    cout << *chPtr;
    chPtr--;
    cout << *chPtr;
```

To:

- Decrement `chPtr` to the previous element in the array.
- Output the value de-referenced by `chPtr`.
- Decrement `chPtr` to the previous element in the array.
- Output the value de-referenced by `chPtr`.

```
Compile and run the program.
```

```
abcba
```

Warning: When using the increment / decrement operators with pointer variables be very careful not to go beyond the boundaries of the array.

```
Close the file.
```

Dynamic Memory

When variables are declared, including arrays with a fixed size, the amount of required memory is allocated prior to program execution. This is what is referred to as static memory. That is, it is fixed, it cannot be varied.

An alternative to allocating memory statically is to allocate it dynamically. That is, during program execution.

Pointer to Variables

Create a program named "`dynamicVariables.cpp`" and add the following code:

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

int main ()
{

   return 0;
}
```

Allocation

In order to allocate memory dynamically, the operator (`new`) is used.

Add the following code to `main`:

```
int * iPtr = NULL;
iPtr = new int;
```

To:

- Declare an integer pointer named `iPtr`.

Allocate memory for one integer and returns its address to `iPtr`.

The operator (**new**) allocates the requested amount of memory and returns the address of its location in memory. Notice that when using `new` the allocated memory does not have a name, as opposed to a statically allocated variable. Therefore, the only way to access the memory is by its address which has been assigned to **`iPtr`**.

Accessing

Now that **`iPtr`** has a valid address associated with it, it can be accessed as usual by de-referencing the pointer.

Add the following code to `main`:

```
*iPtr = 100;
cout << *iPtr << endl;
```

To:

- Assign the value `100` to the memory de-referenced by **`iPtr`** and output its value.

```
Compile and run the program.
```

```
100
```

De-Allocation

When using the operator (**new**) to allocate memory, it is the programmer's responsibility to de-allocate the memory. That is, return the memory to the operating system. The compiler does not do this automatically as with static memory.

If the programmer does not de-allocate the memory assigned dynamically, the program will *leak* memory. This means that the allocated memory has not been released back to the operating system when no longer required and cannot be used by another application. If the size of the leak is large, it could affect operation of the system as more and more memory is consumed. However, when the application which has consumed the memory has terminated the memory is eventually restored to the OS.

In order to de-allocate memory, the operator (**`delete`**) is used.

Add the following code to `main`:

```
delete iPtr;
iPtr = NULL;
```

To:

- De-allocate the memory that `iPtr` references and set `iPtr` to the *null* value.

Note: Remember to set all pointer variables to NULL after deleting memory.

```
Close the file.
```

Pointer to Array

Static memory has a major limitation in that the programmer must declare the amount of memory in the source code. This is not a problem for individual variables but when it relates to arrays the programmer must decide how much memory to allocate. Often, however, it is not possible to know the exact amount of memory required. So, the programmer must make a decision about how much memory to allocate and risks either not allocating enough memory and having the program falter, or allocating an excessive amount of memory and thus being wasteful of resources.

The way to overcome this problem is to allocate the memory for arrays dynamically. That is, during program execution. Either the user enters the required size of the array, or its size is determined from some other method. I.e. The array data is to be populated from a file and the file contains some information indicating the required size.

Create a program named "`dynamicArrays.cpp`" and add the following code:

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

int main ()
{

   return 0;
}
```

Allocation

Allocating memory for arrays is similar to single variables except that the array operator (`[]`) is used as in static array declaration to indicate the size of the memory required.

Add the following code to `main`:

```
int * iArrPtr = NULL;
int size = 0;
cout << "Enter the size of the array: ";
cin >> size;
iArrPtr = new int [size];
```

To:

- Declare an integer pointer variable named `iArrPtr`.
- Declare an integer variable named `size`.
- Receive from user input the value for `size`, which is the size of the array.
- Allocate memory for `size` integers and assign the address of the first memory location to `iArrPtr`.

The operator (`new`) allocates the requested amount of memory, in contiguous locations,

and returns the base address (first location) of the memory. Again, the memory is unnamed and the only way to access the memory is by its address, which `iArrPtr` holds.

Accessing

Now that the memory has been allocated, in order to access each of the five elements, the array subscript (`[]`) operator can be used in exactly the same way as for static arrays. Add the following code to `main`:

```
for (int i = 0; i < size; i++)
   iArrPtr [i] = (i+1)*11;
```

To:

- Set the values in the array referenced by `iArrPtr` to `11, 22... n`.

Alternatively, pointer arithmetic can be used to access the memory. As a pointer contains an address and the memory for an array is allocated in contiguous locations, 1, 2 ... n-1 is added to the first address (stored in `iArrPtr`) to give the address of the next locations.

Remember that `iArrPtr` contains the address of the first element in the array, so `iArrPtr` is equal to `iArrPtr [0]`. Using pointer arithmetic, `iArrPtr + 0` is equal to `iArrPtr` which is equal `iArrPtr[0]`. Thus, `iArrPtr+i` is equivalent to `iArrPtr[i]`.

Add the following code to `main`:

```
for (int i = 0; i < size; i++)
   cout << *iArrPtr+i << " ";
```

To:

- Output each of the values in the array.

```
Compile and run the program.
============================================================
Enter the size of the array: [5]
11 12 13 14 15
```

Notice that the output is not representative of the values assigned to each of the array elements (`11 22 33 44 55`). This is because the operator (`*`) has a higher precedence than (`+`), and the address of `iArrPtr` is de-referenced, retrieving the value `11`, *before* adding (`0 1 2 3 or 4`).

Modify the code:

```
for (int i = 0; i < size; i++)
   cout << *iArrPtr+i << " ";
```

To the following:

```
for (int i = 0; i < size; i++)
   cout << *(iArrPtr+i) << " ";
```

To:

- Produce the *correct* output of the values in each of array elements.

Now, each address is accessed before de-referencing its value.

```
Compile and run the program.
=============================================
Enter the size of the array: [5]
11 22 33 44 55
```

Warning: As with statically declared arrays, be very careful to avoid accessing memory out of the boundaries of the array.

De-Allocation

When dynamically allocated memory is no longer required it must be released back to the operating system. The operator (`delete`) achieves this purpose.

Add the following code to `main`:

```
delete [] iPtr;
iPtr = NULL;
```

To:

- Release the memory referenced by `iPtr` and set `iPtr` to `NULL`.

Notice that the operator (`[]`) is used in the `delete` statement. This is because `iPtr` references an array and must be included.

After releasing memory using delete, the pointer should be set to `NULL` to indicate that it no longer references memory. If more than one pointer references the same memory, all must be set to `NULL`.

Programming Exercise:
Create a program that copies data from one dynamic array to a second dynamic array in reverse order.

1. Create a C++ program named "`dynamicArraysCopy.cpp`".
2. Declare two integer pointer variables named `iArrPtr1` and `iArrPtr2`. and one integer variable named `size`.
3. Receive user input for the `size` of the arrays.

4. Dynamically allocate memory for each of the arrays, based on the user input value from 3. above.
5. Receive user input to populate all of the elements of the array referenced by `iArrPtr1`.
6. Copy all of the elements referenced by `iArrPtr1` to the memory referenced by `iArrPtr2` in reverse order. That is, `iArrPtr2+0` will be equal to `iArrPtr1+(size-1)`. *Note: Use array subscripting to access the array memory.*
7. Output the content of the two arrays to verify that the copy is correct. *Note: Use pointer arithmetic to access the array memory.*
8. Delete the memory allocated to each array.

Array of Pointers

Previously, a pointer to an array of memory (static or dynamic).

Create a program named "`arrayOfPointers.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

An array of pointers can reference either static or dynamic memory.

Add the following code to `main`:

```
int * iPtrArr[5] = {NULL};
int iArray[5] = {0};

for (int i = 0; i < 5; i++)
   iPtrArr[i] = &iArray[i];

for (int i = 0; i < 5; i++)
   *iPtrArr[i] = i+1;

for (int i = 0; i < 5; i++)
   cout << *iPtrArr[i] << " ";
```

To:

- Declare an array of integer pointers named `iPtrArr and set the values of each element to NULL.`
- Declare an array of integer variables named `iArray` and initialise each element to the value 0.
- Iterate, assigning the address of each element in `iArray` to each element in `iPtrArr`.
- Iterate, setting the de-referenced values of `iPtrArr` to { `1 2 3 4 5` }.
- Output the values de-referenced by each pointer in `iPtrArr`.

```
Compile and run the program.
```

```
1 2 3 4 5
```

Add the following code to `main`:

```
for (int i = 0; i < 5; i++)
   cout << iPtrArr[i] << " ";
cout << endl;

for (int i = 0; i < 5; i++)
      cout << *(iPtrArr+i) << " ";
   cout << endl;
```

To:

- Output the values in `iPtrArr[]` (addresses) and the de-referenced values.

```
Note: Pointer arithmetic does not work as expected with an
array of pointers as the memory associated with each pointer
is not necessarily in contiguous locations.
```

Void Pointers

The `void` pointer is a generic pointer that can be used to hold an address but as it has no type or pre-determined length, it cannot be used to de-reference values. However, it can point to any data type, from an integer or float value to a string of characters. `void` pointers are used when the type of the data is not yet determined or when it may vary i.e. as a function parameter.

Create a file named "`pointersVoid.cpp`" and add the following code:

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

int main ()
{
   return 0;
}
```

To:

- Define the `main` function.

Declaring a void pointer and assigning an address is the same as declaring a pointer of any other type.

Add the following code to `main`:

```
int iVar = 10;
void * vPtr;
vPtr = &iVar;
```

To:

- Declare an `int` variable and a `void` pointer (`void*`) variable and assign the address of the `int` to the `void*`.

`vPtr` now has the address of `iVar` as its value, although it does not know its type and subsequent length.

The use of `void` pointers can make the code more accessible however, in exchange there are limitations: the data referenced by them cannot be directly de-referenced. For that reason the type of the `void` pointer will need to be cast to the type of the memory it references.

Add the following code to `main`:

```
cout << "The value de-referenced by vPtr is "
   << *(static_cast<int*>(vPtr)) << endl;
```

To:

- Output the value de-referenced by `vPtr`.

Before the `void` pointer can be de-referenced, it first needs to be cast to a pointer type of the memory it references so that the compiler knows how to interpret the data. In this case, it is `int*`.

```
Compile and run the program.
============================================================
The value de-referenced by vPtr is 10
```

Likewise, the value can be altered by accessing the memory in the same way.

Add the following code to `main`:

```
*(static_cast<int*>(vPtr)) += 1;
cout << "The value in de-referenced by vPtr is "
   << *(static_cast<int*>(vPtr)) << endl;
```

To:

- Increment the value de-referenced by `vPtr` by one and output the value.

```
Compile and run the program.
```

```
The value in de-referenced by vPtr is 10
The value in de-referenced by vPtr is 11
```

Void Pointers as Function Parameters

A void pointer can be passed to a function like any other parameter.

Add the following prototype *before* `main`:

```
int cmpInt (void*, void*);
```

To:

- Declare the function `cmpInt` that takes two `void*` parameters.

Add the following definition *after* `main`:

```
int cmpInt (void* a, void* b)
{
   return *(static_cast<int*>(a)) - *(static_cast<int*>(b));
}
```

To:

- Define the function `cmpInt` that takes two `void*` parameters, compares their de-referenced values and returns the difference between the two.

Add the following code to `main`:

```
int i1, i2;

cout << "Enter two integers: " ;
cin >> i1 >> i2;

int diff = cmpInt (&i1, &i2);

if (diff < 0)
   cout << i1 << "  <  " << i2 << endl;
else if (diff > 0)
   cout << i1 << "  >  " << i2 << endl;
else
   cout << i1 << " == " << i2 << endl;
```

To:

- Declare two `int` variables, receive user input for their values and pass their addresses to `cmpInt` for comparison and output accordingly.

```
Compile and run the program.
```

```
Enter two integers: [33 44]
33  <  44
```

Add the following prototype *before* `main`:

```
int cmpFloat (void*, void*);
```

To:

- Declare the function `cmpInt()` that takes two `void*` parameters.

Add the following definition *after* `main`:

```
int cmpFloat (void* a, void* b)
{
   if (*(static_cast<float*>(a)) < *(static_cast<float*>(b)))
       return -1;
   else if (*(static_cast<float*>(a)) >
             *(static_cast<float*>(b)))
       return 1;
   else
       return 0;
}
```

To:

- Define the function `cmpFloat()` that takes two `void*` parameters, compares their de-referenced values and returns the difference between the two.

Add the following code to `main`:

```
float f1, f2;
void * vPtr1, *vPtr2;

cout << "Enter two floats: " ;
cin >> f1 >> f2;

vPtr1 = &f1;
vPtr2 = &f2;
diff = cmpFloat (vPtr1, vPtr2);

if (diff < 0)
   cout << f1 << "  <  " << f2 << endl;
else if (diff > 0)
   cout << f1 << "  >  " << f2 << endl;
else
   cout << f1 << " == " << f2 << endl;
```

To:

- Declare two `float` variables, receive user input for their values, assign their addresses to two `void*` variables and pass them to `cmpFloat()` for comparison and output accordingly.

As the two parameter variables to `cmpFloat()` are `void*`, the arguments passed to the function can be either addresses of variables *or* `void*` variables.

Compile and run the program.

```
Enter two integers: [34 56]
34  <  56
Enter two floats: [45.3 76.2]
45.3  <  76.2
```

Pointers to Functions

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function, since these cannot be passed de-referenced. In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses `()` and an asterisk (`*`) is inserted before the name:

Add the following code *to the top* of `main`:

```
int (*fptr)(void*, void*);
fptr = cmpInt;
```

To:

- Declare the function pointer, `fptr` that takes two `void*` parameters and returns an `int`.
- Assign the function `cmpInt` to the function pointer `fptr`.

Declaring a function pointer is very similar to declaring a standard function except the function name is encased in `()` and the pointer (`*`) operator is included.

Any function that is assigned to a function pointer must have the same signature. This consists of the return type and parameter type(s).

Notice also that in the assignment statement "`function pointer = function;`" `i.e. fptr = cmpInt;` the `()` are not included.

Now, instead of invoking the function `cmpInt` directly, the function can be called indirectly using the function pointer, `fptr`.

Replace the code:

```
int diff = cmpInt (&i1, &i2);
```

With:

```
int diff = fptr (&i1, &i2);
```

And the code:

```
diff = cmpFloat (vPtr1, vPtr2);
```

With:

```
fptr = cmpFloat;
diff = fptr (vPtr1, vPtr2);
```

To:

- To replace the two function calls to `cmpInt` and `cmpFloat` with calls using the function pointer, `fptr`.

The code in `main` should now resemble:

```
int main ()
{
   int (*fptr)(void*, void*);
   fptr = cmpInt;

   int i1, i2;

   cout << "Enter two integers: " ;
   cin >> i1 >> i2;

   int diff = fptr (&i1, &i2);

   if (diff < 0)
      cout << i1 << "  <  " << i2 << endl;
   else if (diff > 0)
      cout << i1 << "  >  " << i2 << endl;
   else
      cout << i1 << " == " << i2 << endl;

   float f1, f2;
   void * vPtr1, *vPtr2;

   cout << "Enter two floats: " ;
   cin >> f1 >> f2;
   vPtr1 = &f1;
   vPtr2 = &f2;

   fptr = cmpFloat;
   diff = fptr (vPtr1, vPtr2);

   if (diff < 0)
      cout << f1 << "  <  " << f2 << endl;
   else if (diff > 0)
      cout << f1 << "  >  " << f2 << endl;
   else
      cout << f1 << " == " << f2 << endl;
   return 0;
}
```

```
Compile and run the program.
=============================================================
Enter two integers: [67 45]
67  >  45
Enter two floats: [34.5 77.7]
34.5  <  77.7
```

Function pointers are most commonly used as a parameter to another function.

Add the following function prototype *before* `main`:

```
int cmpVal(void * a, void* b, int(*fptr)(void*, void*))
```

To:

- Declare the function `cmpVal` that returns an `int` and takes two `void*` parameters and one function pointer parameter.

The parameter declaration for the function pointer is exactly the same as would be declared in `main`, for example.

Add the following function definition *after* `main`:

```
int cmpVal(void * a, void* b, int(*fptr)(void*, void*))
{
   return fptr (a, b);
}
```

The intent here is that `cmpVal` is a generic function that can take any type for the parameters `a` and `b` and any function that has the same signature as `fptr` for the parameter `fptr`. It in turn calls the function referenced by `fptr` passing the other two parameters.

Replace the two lines of code:

```
int diff = fptr (&i1, &i2);
AND
diff = fptr (vPtr1, vPtr2);
```

With:

```
int diff = cmpVal (&i1, &i2, cmpInt);
AND
diff = cmpVal (vPtr1, vPtr2, cmpFloat);
```

To:

- Call the function `cmpVal`, passing the addresses of `i1` and `i2` along with the function `cmpInt`.
- Call the function `cmpVal`, passing the two `void*` variables which reference `f1` and `f2` along with the function `cmpFloat`.

Compile and run the program.

```
Enter two integers: [100 100]
100 == 100
Enter two floats: [34 99.9]
<  99.9
```

Close the file.