C++ Mini Lessons Last Update: Mar 31, 2021

From http://www.onlineprogramminglessons.com

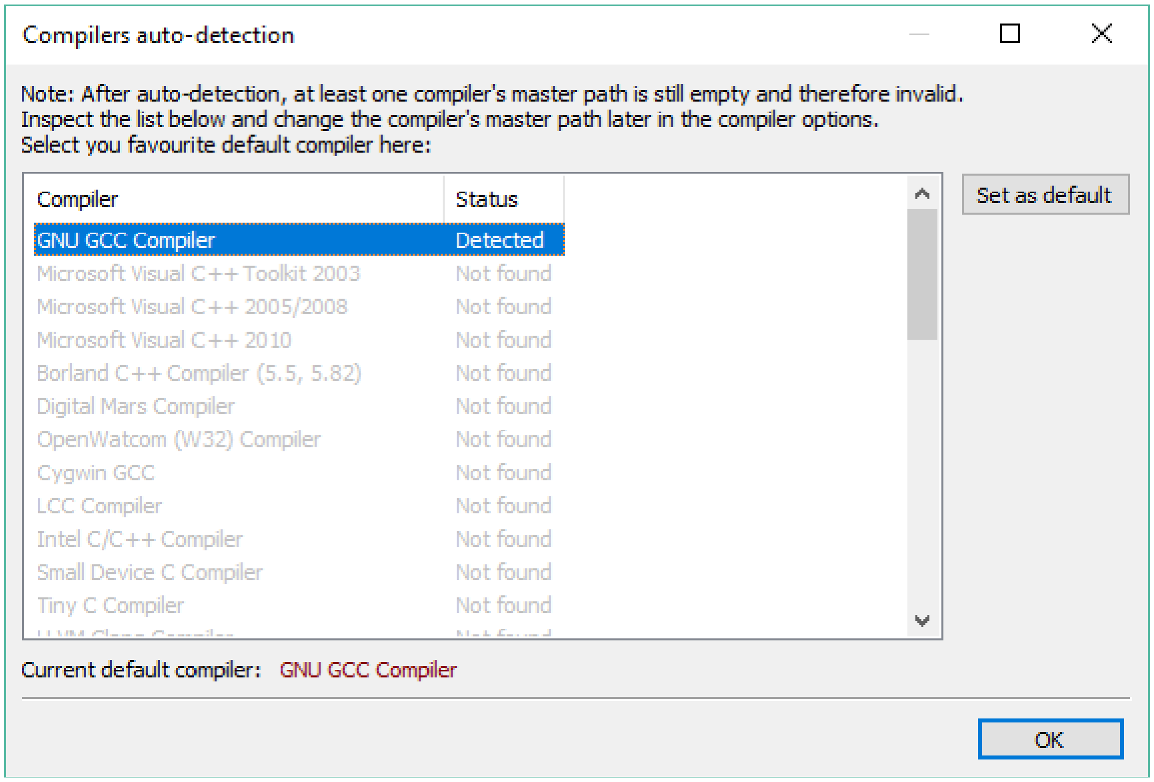
These C++ mini lessons will teach you all the C++ Programming statements you need to know, so you can write 90% of any C++ Program.

Let's get started!
You first need to download a C++ Compiler. This is needed to edit, compile and run C++ programs. You can download the Codeblocks C++ Compiler or alternately you can use your own C++ compiler. Download CodeBlocks from this link.

http://www.codeblocks.org/downloads

Download and install the **codeblocks-17.12mingw-setup.exe** file that includes everything

Once you installed and run CodeBlocks you will get this screen that asks you to select the C++ Compiler to use. We selected the GNU GCC Compiler and then pressed the "Set as Default" button.

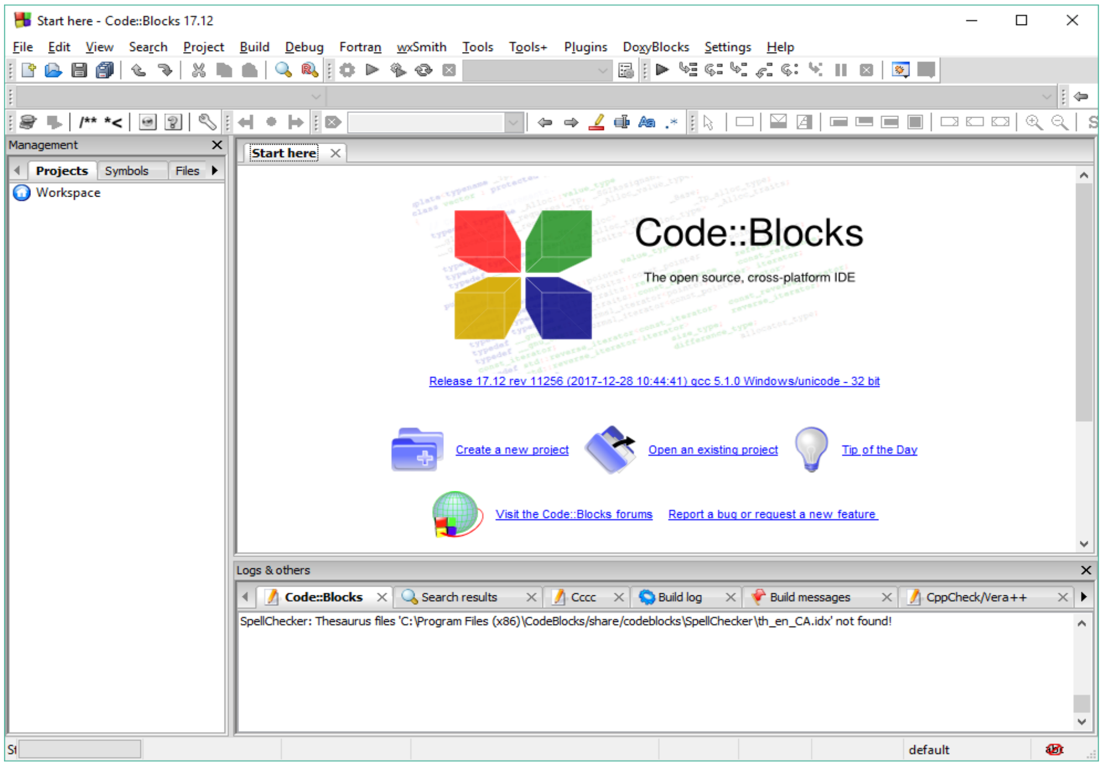

Compilers auto-detection

Note: After auto-detection, at least one compiler's master path is still empty and therefore invalid.
Inspect the list below and change the compiler's master path later in the compiler options.
Select you favourite default compiler here:

Compiler	Status
GNU GCC Compiler	Detected
Microsoft Visual C++ Toolkit 2003	Not found
Microsoft Visual C++ 2005/2008	Not found
Microsoft Visual C++ 2010	Not found
Borland C++ Compiler (5.5, 5.82)	Not found
Digital Mars Compiler	Not found
OpenWatcom (W32) Compiler	Not found
Cygwin GCC	Not found
LCC Compiler	Not found
Intel C/C++ Compiler	Not found
Small Device C Compiler	Not found
Tiny C Compiler	Not found

Current default compiler: GNU GCC Compiler

The following screen then appears.


Start here - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Management
Projects Symbols Files
Workspace
Start here
Code::Blocks
The open source, cross-platform IDE
Release 17.12 rev 11256 (2017-12-28 10:44:41) gcc 5.1.0 Windows/unicode - 32 bit
Create a new project
Open an existing project
Tip of the Day
Visit the Code::Blocks forums
Report a bug or request a new feature
Logs & others
Code::Blocks
Search results
Cccc
Build log
Build messages
CppCheck/Vera++
default

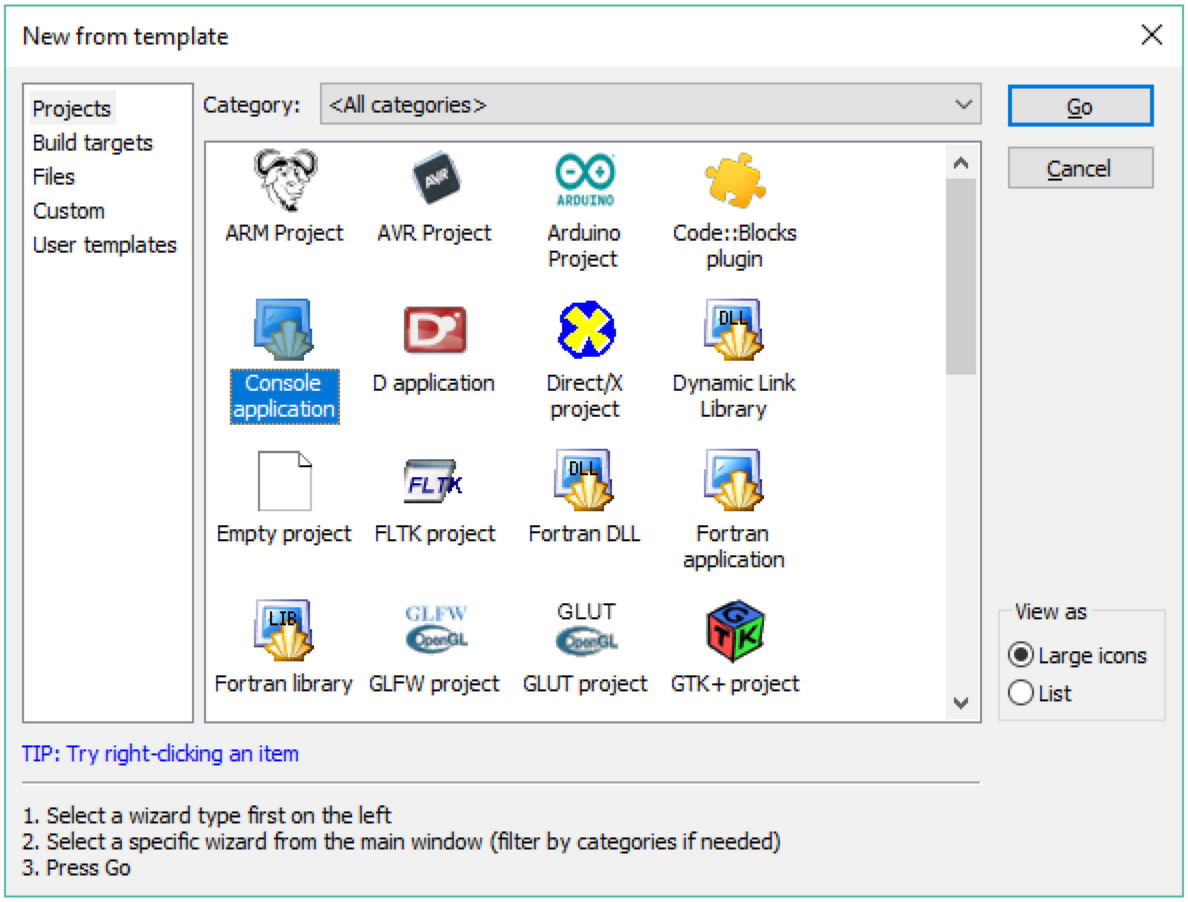
```
SpellChecker: Thesaurus files 'C:\Program Files (x86)\CodeBlocks/share/codeblocks\SpellChecker\th_en_CA.idx' not found!
```

<u>Lesson 1</u> Input and Output

You first need to make a Project to store all your C++ lesson files.

From the start menu select **Create a New Project** or from the file menu select, **Open New Project**.

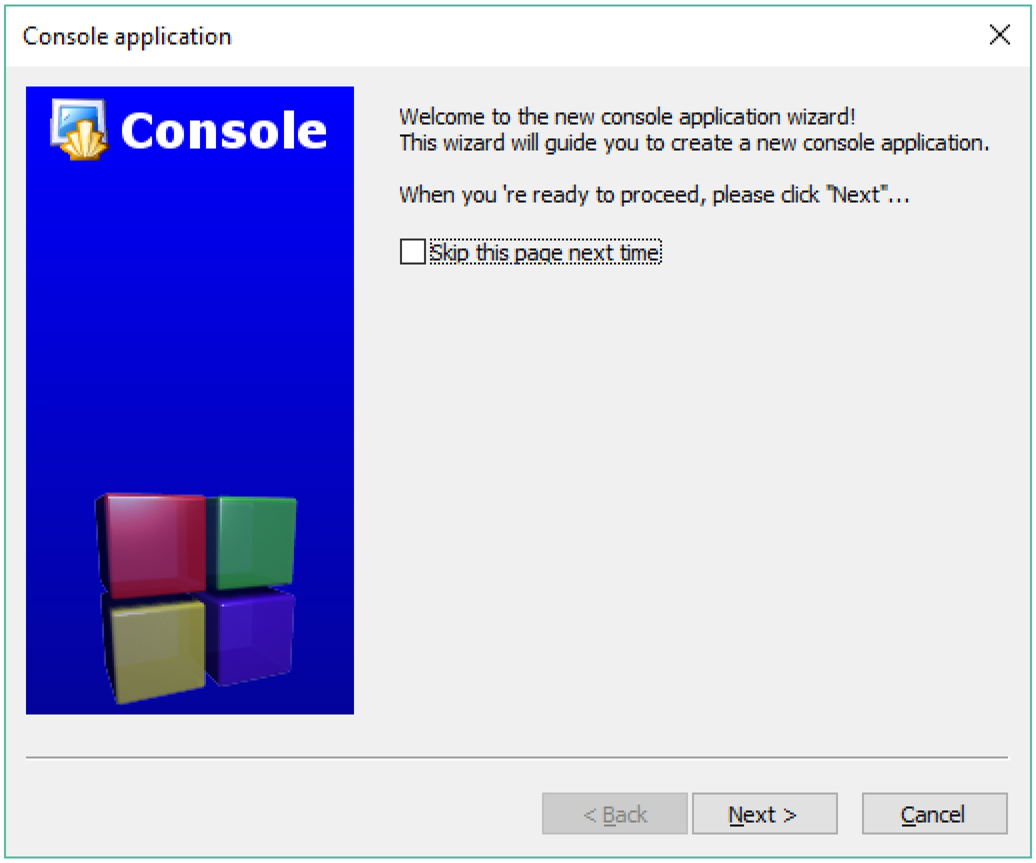
Next Select **Console application** and then press the Go button.



TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

The Console screen appears next.

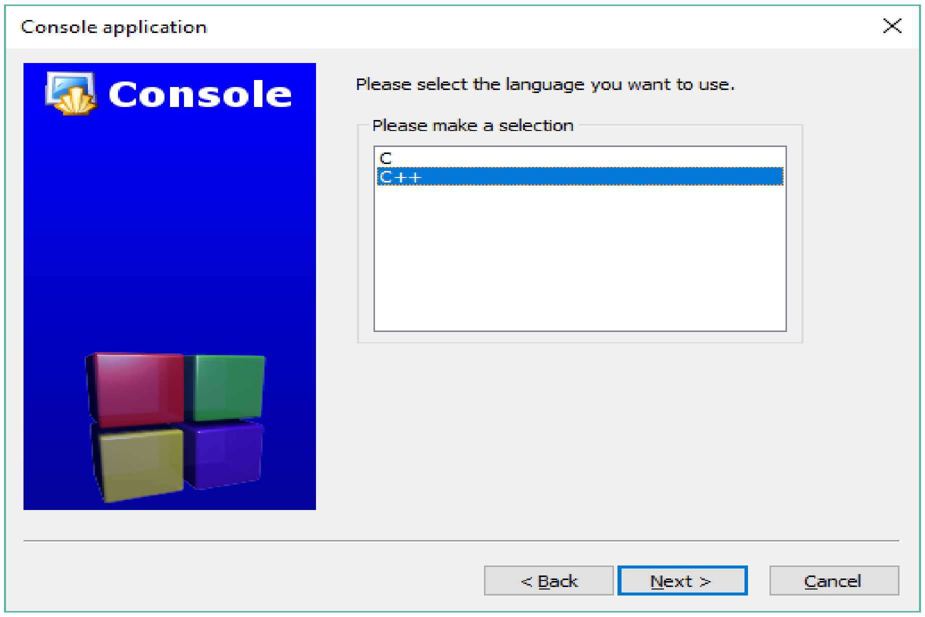

Console application

Welcome to the new console application wizard!
This wizard will guide you to create a new console application.

When you 're ready to proceed, please click "Next"...

Skip this page next time

Press Next button


Console application

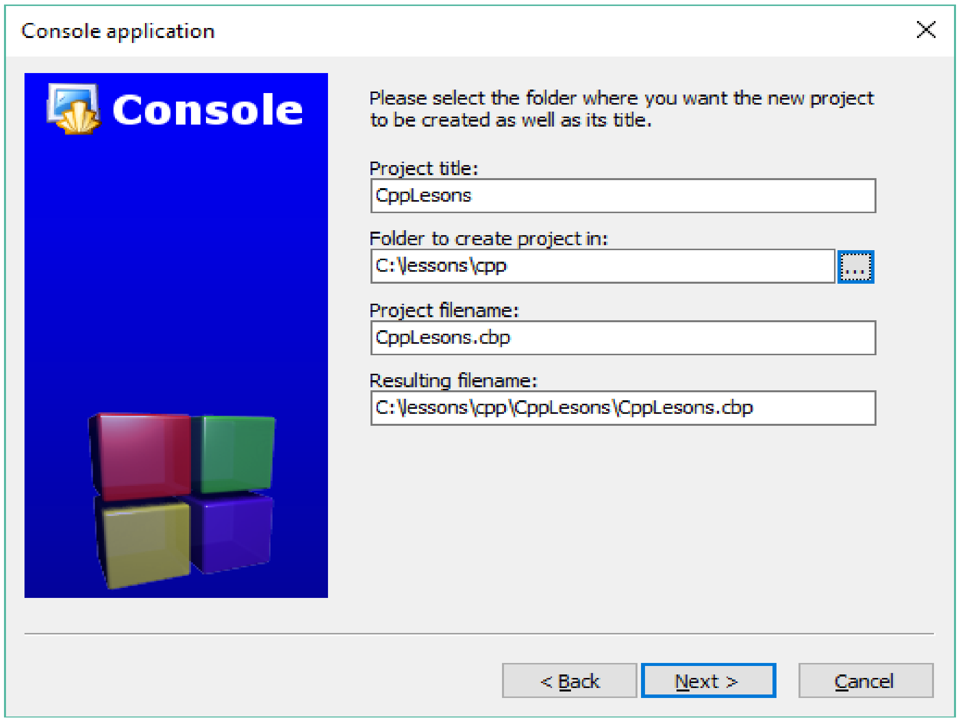
Please select the language you want to use.

Please make a selection

C
C++

Select C++ and then press Next button

Using File Explorer make a Folder to hold your C++ lesson files called Cpp, and then enter CppLessons as the project name.



Please select the folder where you want the new project to be created as well as its title.

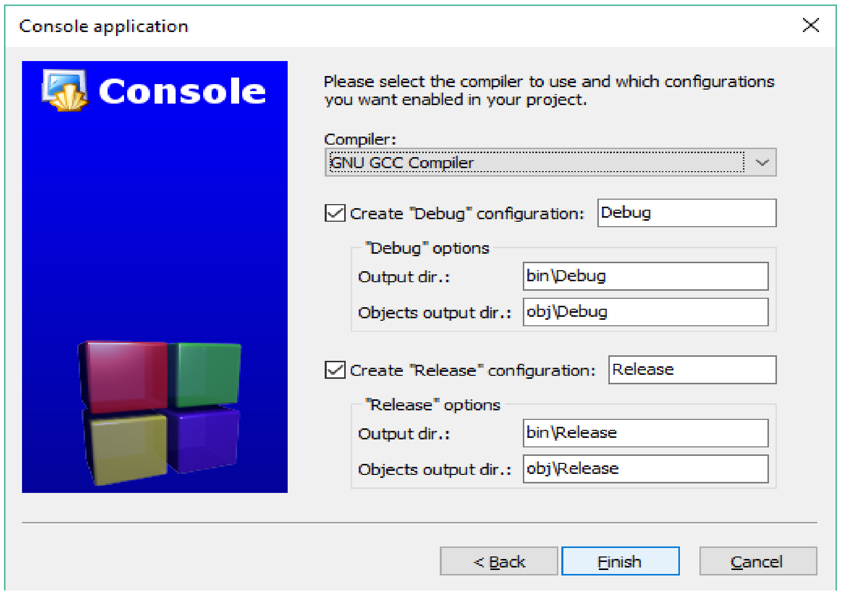
Project title: CppLesons

Folder to create project in: C:\lessons\cpp

Project filename: CppLesons.cbp

Resulting filename: C:\lessons\cpp\CppLesons\CppLesons.cbp

The next screen tells you what C++ Complier the CodeBlocks is using.



Please select the compiler to use and which configurations you want enabled in your project.

Compiler: GNU GCC Compiler

☑ Create "Debug" configuration: Debug

"Debug" options

Output dir.: bin\Debug

Objects output dir.: obj\Debug

☑ Create "Release" configuration: Release

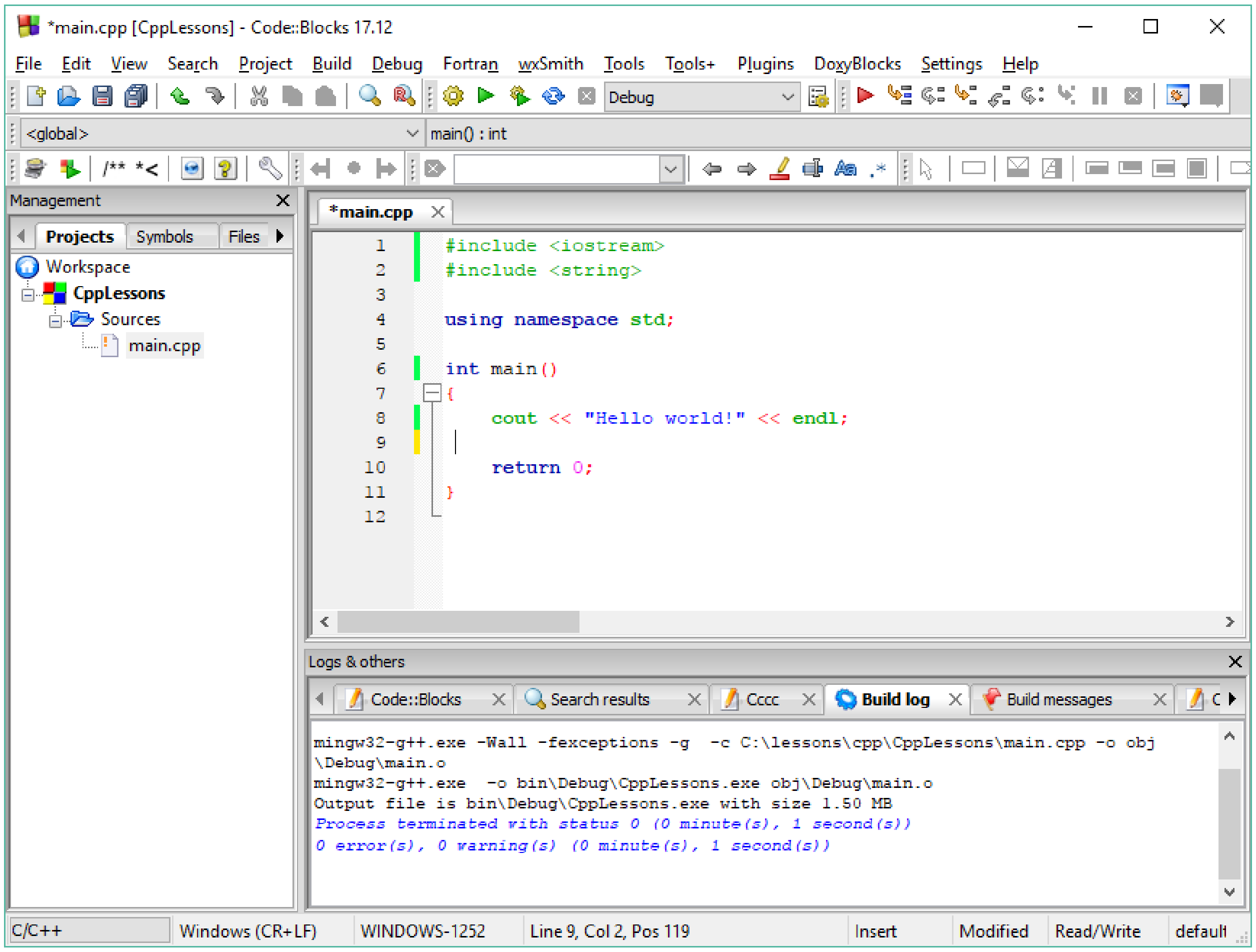
"Release" options

Output dir.: bin\Release

Objects output dir.: obj\Release

You can just select Finish

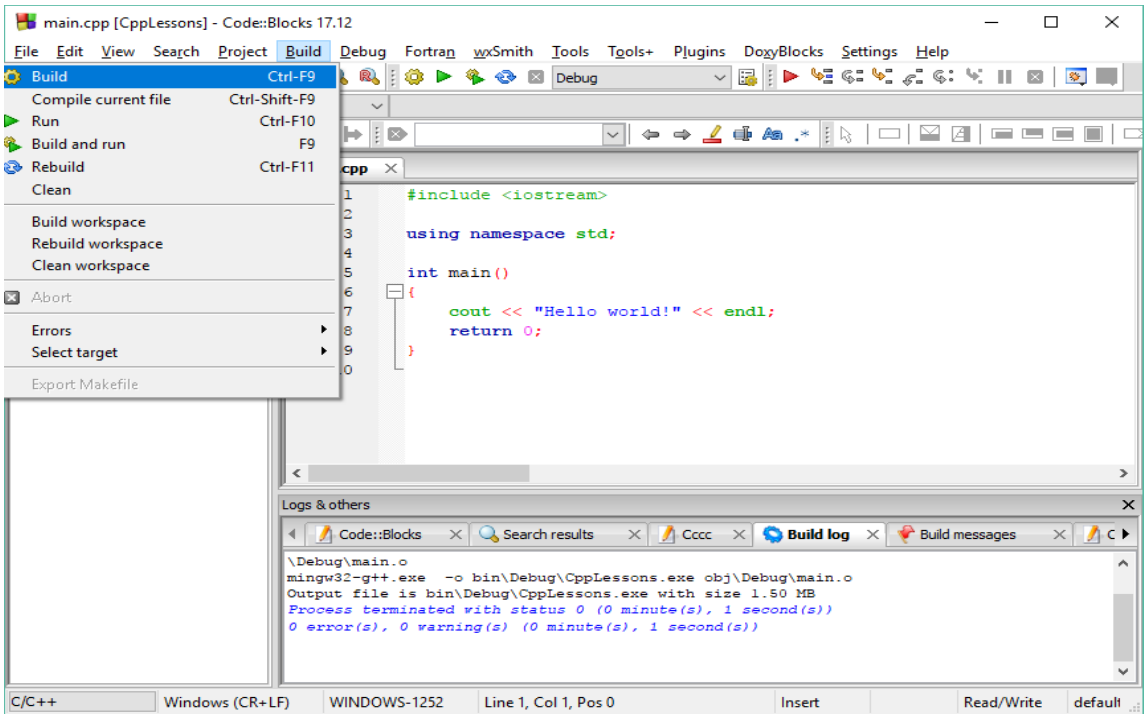
You should get something like this after clicking on main.cpp in the Management window.


*main.cpp [CppLessons] - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
<global>
main() : int
Management
Projects Symbols Files
Workspace
CppLessons
Sources
main.cpp
*main.cpp
Logs & others
Code::Blocks Search results Cccc Build log Build messages
C/C++ Windows (CR+LF) WINDOWS-1252 Line 9, Col 2, Pos 119 Insert Modified Read/Write default

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

using namespace std;

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

    return 0;
}

```

```
mingw32-g++.exe -Wall -fexceptions -g  -c C:\lessons\cpp\CppLessons\main.cpp -o obj
\Debug\main.o
mingw32-g++.exe  -o bin\Debug\CppLessons.exe obj\Debug\main.o
Output file is bin\Debug\CppLessons.exe with size 1.50 MB
Process terminated with status 0 (0 minute(s), 1 second(s))
0 error(s), 0 warning(s) (0 minute(s), 1 second(s))
```

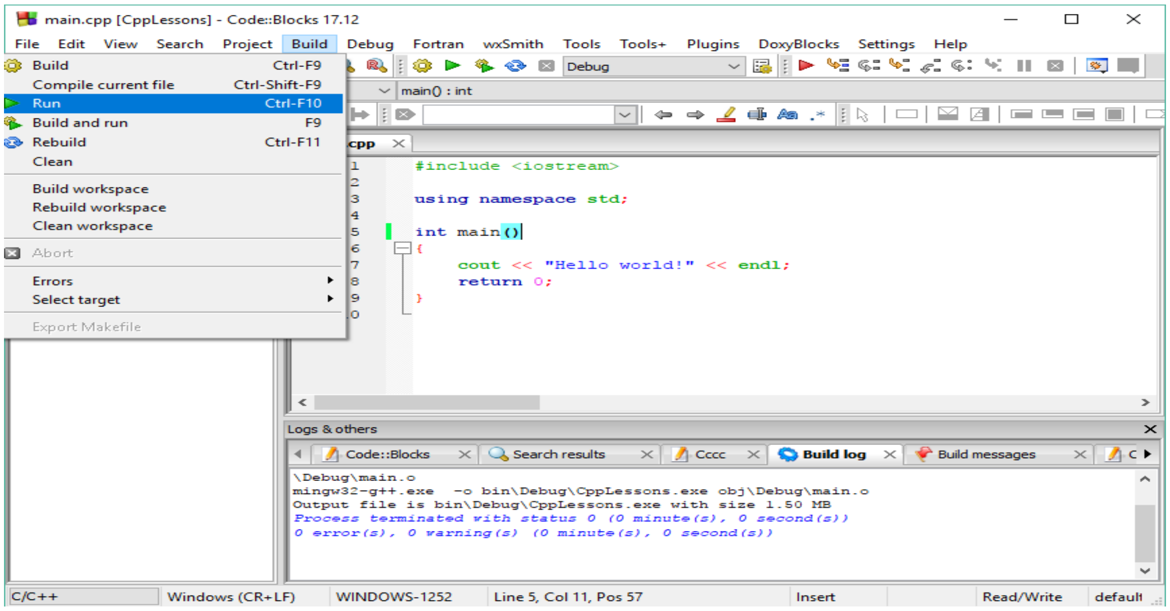
Next you need to build the program before you can run it. From the Build menu select Build.



```
#include <iostream>

using namespace std;

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

```
\Debug\main.o
mingw32-g++.exe  -o bin\Debug\CppLessons.exe obj\Debug\main.o
Output file is bin\Debug\CppLessons.exe with size 1.50 MB
Process terminated with status 0 (0 minute(s), 1 second(s))
0 error(s), 0 warning(s) (0 minute(s), 1 second(s))
```

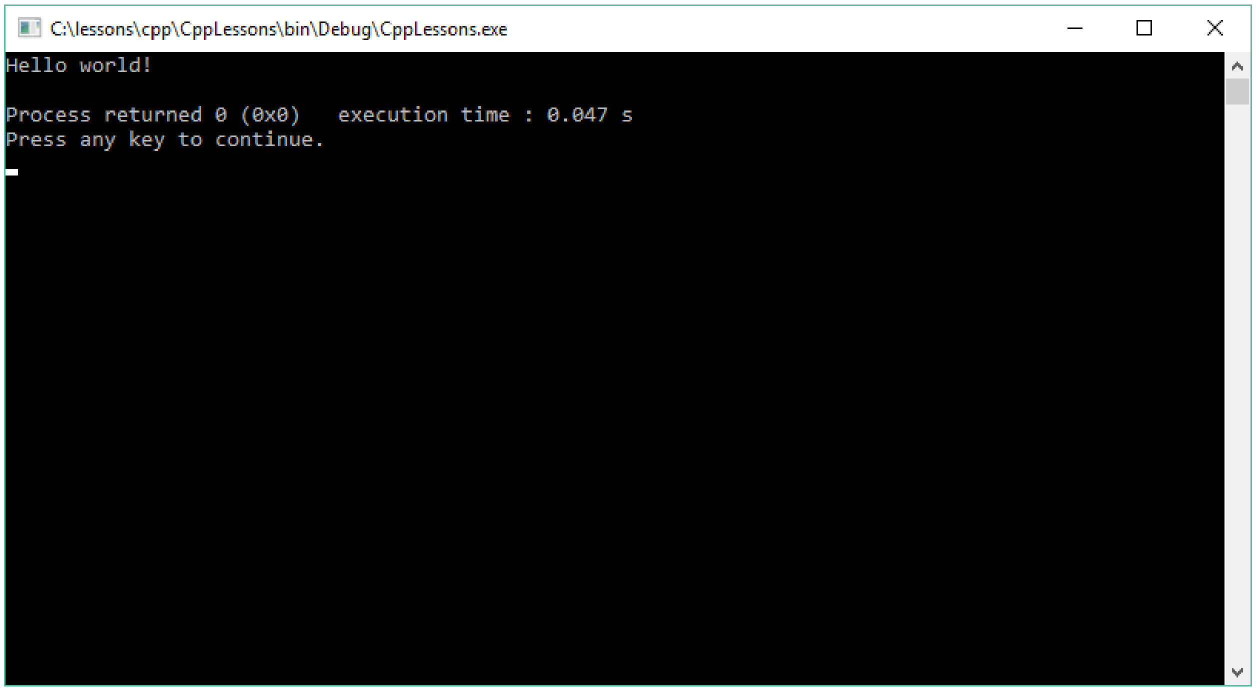
If you have no errors, then you can run your program.
Select Run from the Build menu



```
#include <iostream>

using namespace std;

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

```
\Debug\main.o
mingw32-g++.exe  -o bin\Debug\CppLessons.exe obj\Debug\main.o
Output file is bin\Debug\CppLessons.exe with size 1.50 MB
Process terminated with status 0 (0 minute(s), 0 second(s))
0 error(s), 0 warning(s) (0 minute(s), 0 second(s))
```

You should get something like this:


C:\lessons\cpp\CppLessons\bin\Debug\CppLessons.exe

```
Hello world!

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

A C++ program contains programing statements to tell the computer what to do. These programing statements are grouped together in a function enclosed by curly brackets, so they can run one by one sequentially.

In a C++ program the main function is the first function to run. Here is the program and main function again.

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

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

main is the function name and **cout << "Hello world!" << endl**; is a programming statement printing Hello world! on the screen. All programing statements end in a semicolon ;

Our first C++ program prints our “Hello World” on the screen. If your compiler did not generate the sample program above for you then you will have to type in the programming statemnts your self into your C++ compiler then build and run it.

A C++ program starts with the folowing include statement

```
#include <iostream>
```

The include statement is called a **preprocessor** it tells the C++ compiler to use functions from the iostream file. These functions are actually grouped inside a class definition called **iostream**. When a class definition is used in a program then it becomes an **object**. We will study classes and objects in the preceding lessons so do not worry about them now.

The iostream class belong to the **namespace** std so we need to tell the compiler thate we are using classes from the **std** namespace.

```
using namespace std;
```

The next programing statement is the main functon definition header.

```
int main()
{
```

Inside the main function definition header we have the programming statements enclosed in curly brackets { } The open { curly bracket means to start the programming statements. The closing } curly bracket means to end the programing statements inside the function.

Our first programing statement prints “Hello World” on the console screen

```
cout << "I  like C++ Programming << endl;
```

cout is used to print out messages to the screen. The << operator introduces what is to be printed. The **endl** directive is used to start a new line. **cout** is actially a **ostream** object that is used for output and the << operator is a function of the ostream class that tell’s the program to print messages on the screen.

This main function return's a value The **return** keyword is used to specify what value is to be returned. The main function usually returns a 0 which means every thing is okay.

```
        return 0;
        }
```

Before we proceed it is important to understand the terminology: classes, functions, programming statements and objects.

data type	What type of data is to be represented
variable	Stores a string or numeric value. All variables have a data type
programming statement	is an instruction containing commands to perform a desired action, like printing a value on the screen, get a value from the key board or calculate a certain value
function	Contains programing statements that tell the computer what to do and performs operations on variables using these programming statements
class	Contains values and functions
namespace	contains classes
object	Computer Memory allocated for a class variables when a program runs on the computer

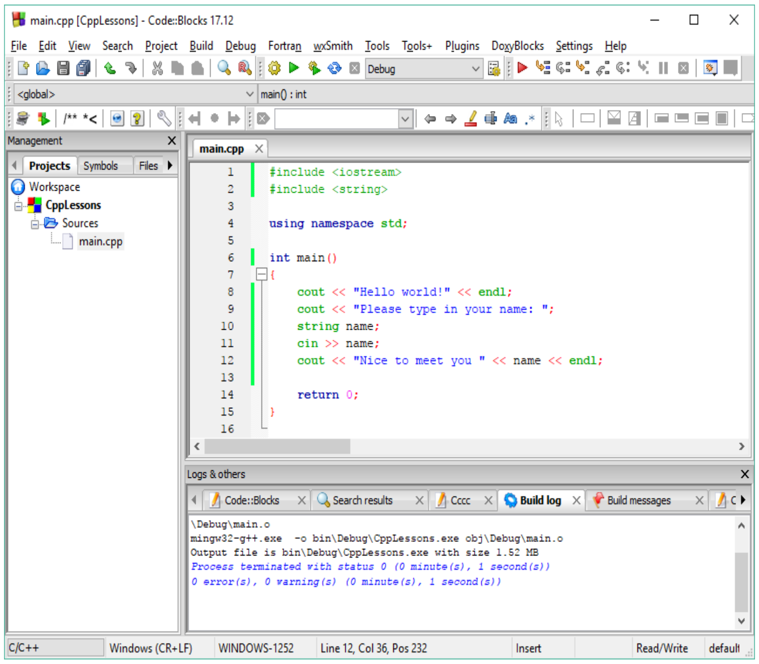
The next thing we need to do is get values from the keyboard. We will ask the user to type in their name and then greet them. Type in the following statements in the C++ editor right after the I like C++ Programming statement.

```
cout << "Please type in your name: ";
string name;
 cin >> name;
cout << "Nice to meet you " << name << endl;
```

You need to put the **#include <string>** preprocessor statement at the top of your program right after **#include <iostream>** so that the compiler knows about string variables. You should have something like this:

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

int main()
{
cout << "I  like C++ Programming" << endl;

cout << "Please type in your name: ";
string name;
 cin >> name;
cout << "Nice to meet you " << name << endl;

return 0;
 }
```

Your codeblocks will now look like this:

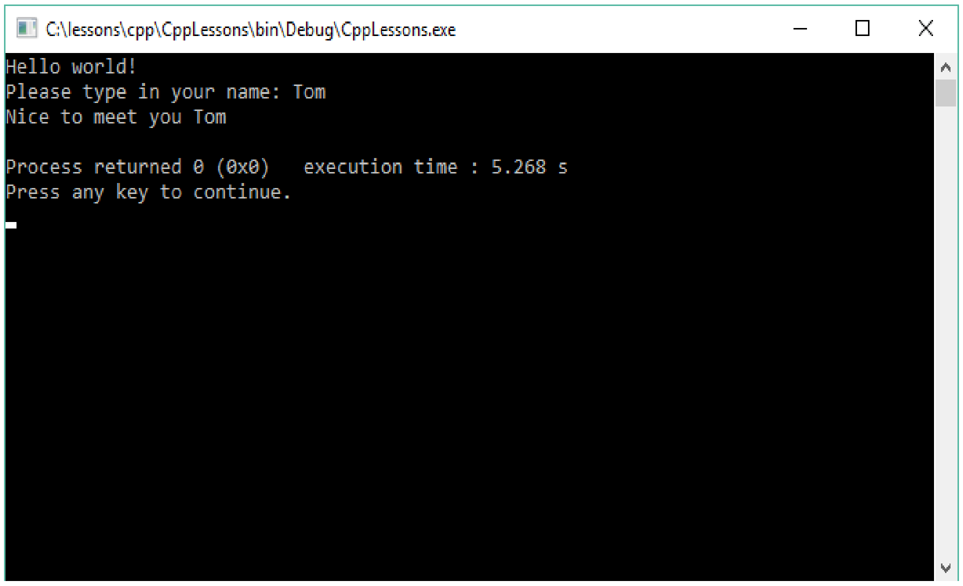


main.cpp [CppLessons] - Code::Blocks 17.12

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

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    cout << "Please type in your name: ";
    string name;
    cin >> name;
    cout << "Nice to meet you " << name << endl;

    return 0;
}
```

```
\Debug\main.o
mingw32-g++.exe  -o bin\Debug\CppLessons.exe obj\Debug\main.o
Output file is bin\Debug\CppLessons.exe with size 1.52 MB
Process terminated with status 0 (0 minute(s), 1 second(s))
0 error(s), 0 warning(s) (0 minute(s), 1 second(s))
```

Now build and run your program, and enter the name “Tom”. You will get something like this:


C:\lessons\cpp\CppLessons\bin\Debug\CppLessons.exe

```
Hello world!
Please type in your name: Tom
Nice to meet you Tom

Process returned 0 (0x0)   execution time : 5.268 s
Press any key to continue.
```

We first ask the user to type in their name using the **cout** statement.

```
cout << "Please type in your name: ";
```

Then we obtain the user name from the keyboard. We first declare a string variable called name and then use **cin** and **operator >>** to read a string from the keyboard**.** The entered name is placed in the string variable name. Variables are used to store values and must be declared before using them.

```
string name;
 cin >> name;
```

cin is actially a **istream** object that is used for input and the >> operator is a function of the istream class used to obtain values from the keyboard.

cout is then used to print out the string message “Nice to meet you" and the name of the user that was stored previously in the variable name.

```
cout << "Nice to meet you " << name << endl;
```

C++ has two types of values: **string** values and **numeric** values. String values are messages enclosed in double quotes like "Hello World" where as numeric values are numbers like 5 and 10.5 Numeric values without decimal points like 5 are known as an **int** and numbers with decimal points like 10.5 are known as a **float** or **double**. Variable's store string or numeric values that can be used later in your program.

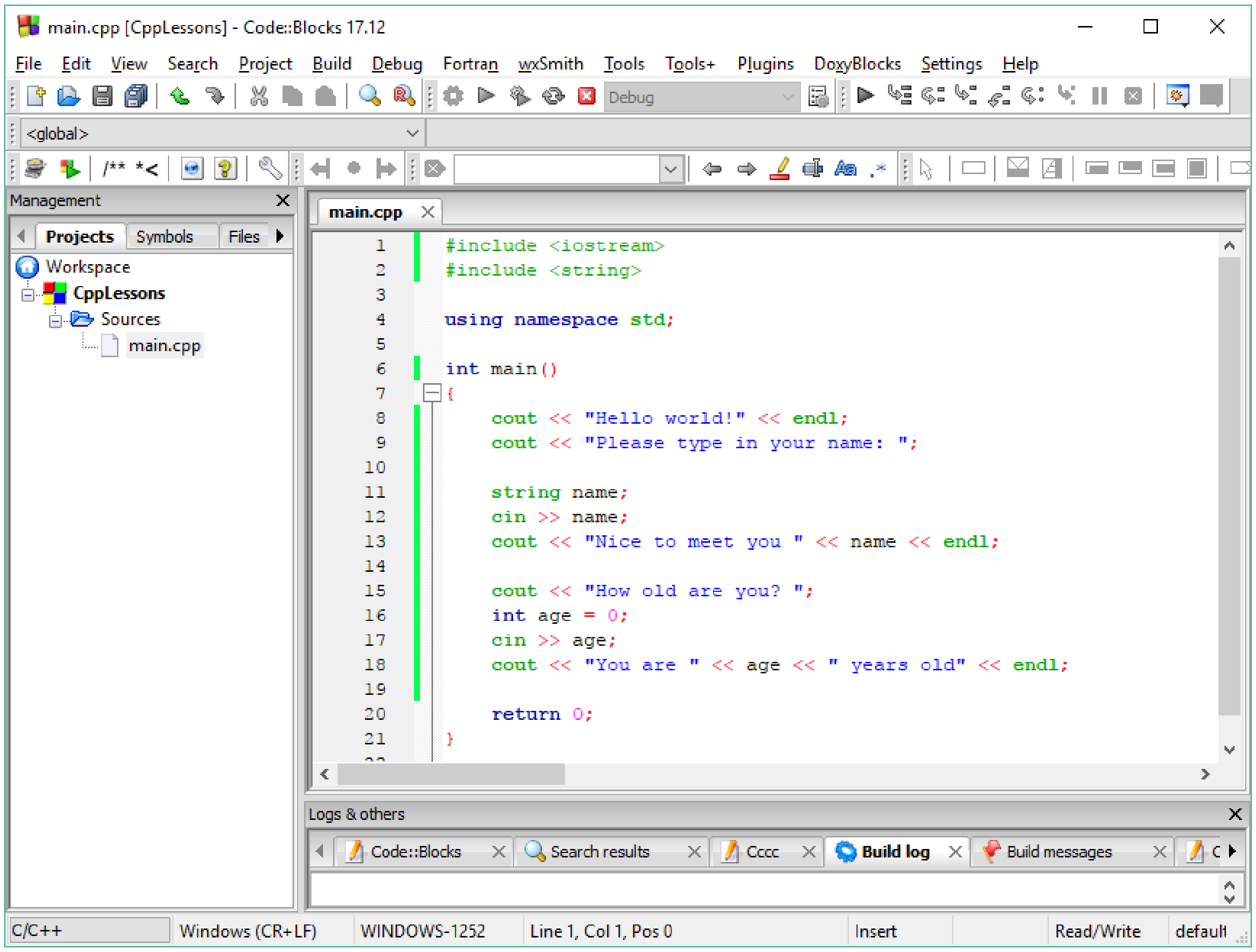
We now continue our program ask the user how old they are. Type in the following statements at the end of your program.

```
cout << "How old are you? ";
int age = 0;
cin >> age;
cout << "You are " << age << " years old" << endl
```

Make sure you save your file before proceeding.
You should have something like this:

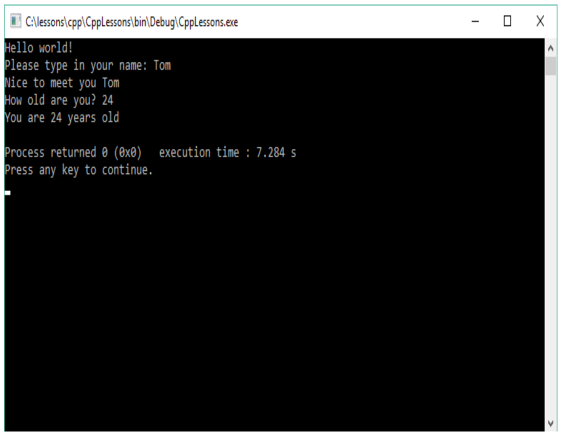
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
cout << "I  like C++ Programming" << endl;

cout << "Please type in your name: ";
string name;
 cin >> name;
cout << "Nice to meet you " << name << endl;
 cout << "How old are you? ";
 int age = 0;
 cin >> age;
  cout << "You are " << age << " years old" << endl

return 0;
```

Your codeblocks will now look like this:


main.cpp [CppLessons] - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Management
Projects Symbols Files
Workspace
CppLessons
Sources
main.cpp
Logs & others
Code::Blocks Search results Cccc Build log Build messages
C/C++ Windows (CR+LF) WINDOWS-1252 Line 1, Col 1, Pos 0 Insert Read/Write default

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

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    cout << "Please type in your name: ";

    string name;
    cin >> name;
    cout << "Nice to meet you " << name << endl;

    cout << "How old are you? ";
    int age = 0;
    cin >> age;
    cout << "You are " << age << " years old" << endl;

    return 0;
}
```

Build and run the program and enter Tom for name and 24 for age, you should get something like this.



```
Hello world!
Please type in your name: Tom
Nice to meet you Tom
How old are you? 24
You are 24 years old

Process returned 0 (0x0)   execution time : 7.284 s
Press any key to continue.
```

We first ask the user to enter there age using cout

```
cout << "How old are you? ";
```

then we declare an **int** variable **age**

```
int age = 0;
```

We enter the age from the keyboard and assign it to the variable age

```
cin >> age
```

Using **cout** we print out the message “You are”, the value store in the variable age and the message “years old”

```
cout << "You are " << age << " years old" << endl;
```

If you have got this far then you will be a great C++ programmer soon.

Most people find Programing difficult to learn. The secret of learning program is to figure out what you need to do and then choose the right programming statement to use. If you want to print messages and values to the screen you use a **cout** statement. If you want to get values from the keyboard, you use a **cin** statement.

If you want to store values, you use variables. You need to tell the compiler what kind of data you want to store, like **int**, **double** or **string**. String is a class, and the string variable is an object. A class is a user specified data type that can do many things that you will find about soon.

You should concentrate on getting your programs running rather than understand how they work. Once you get your programs running and you execute them understanding will be come much easier. Understanding will be much easier, because you can now make an “association connection” to the program statement that is running that produces the desired input or output action.

C++ Data Types

C++ has many data types that can be used to represent different kinds of numbers as follows:

Data Type	Size	Min value	Max Value	Example
char	8	-128	127	byte x = 100;
short	16	-32768	32767	short x = 1000;
int	32	-2^31	2^31-1	int x = 10000;
long	32	-2^31	2^31-1	long x = 10000;
float	32	-1.4E-45	3.4028235E38	float f = 10.5f; (f means float number)
double	64	-4.9E-324	4.9E-324	double d = 10.5;
bool	1	false	True	bool x = true;

The above data types represent signed numbers meaning they can store negative or positive numbers.
bool represents a data type that store true or false values. True is represented by a 1 and false is represented by a 0.
|
Note: C++ also has unsigned data types **unsigned char**, **unsigned short**, **unsigned int** and **unsigned long.**

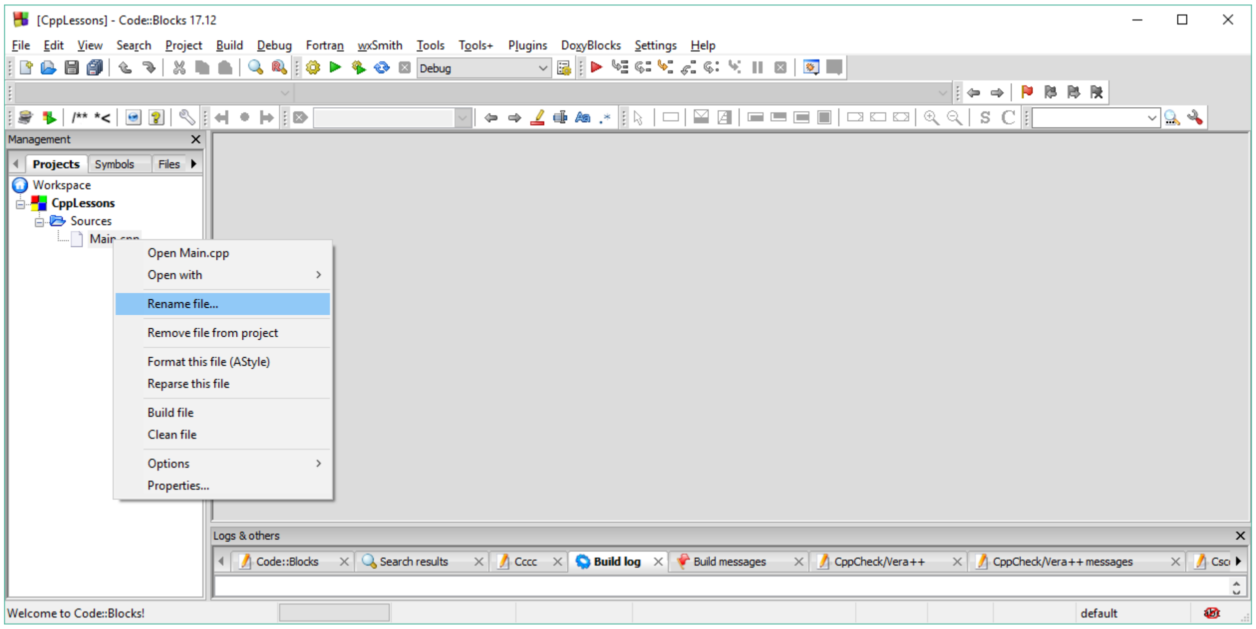
Data Type	Size	Min value	Max Value	Example
uchar	8	0	256	byte x = 100;
ushort	16	0	65535	short x = 1000;
uint	32	0	2^32-1	int x = 10000;
ulong	32	0	2^32-1	long x = 10000;

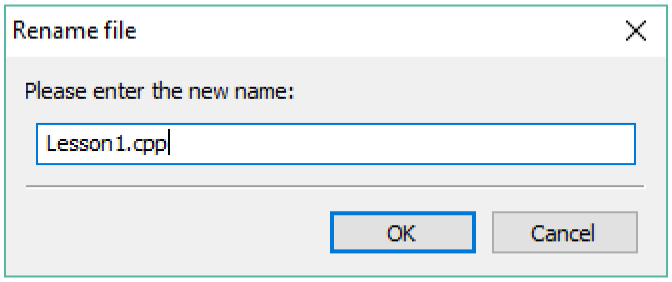
C++ Lesson 1 Homework

Make a C++ program file called homework1.cpp that asks someone what their profession title is and annual salary is. Use a double data type for salary and a string for job title. Print out a message like this: "I am a Manager and I make $100,000 dollars per year!". Also when the program starts print out a welcome message. Put your homework1.cpp file in a C++ project called Homework1.

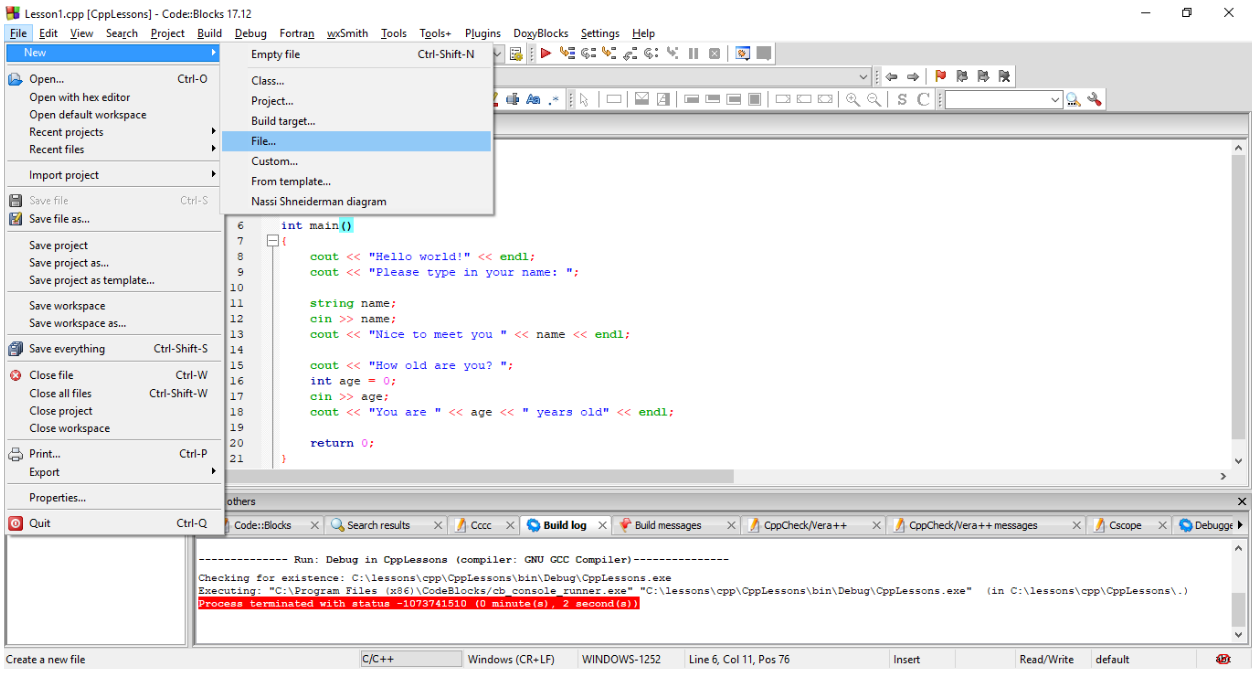
LESSON 2 FUNCTIONS

Functions allow you to group many programming statements together so that you can reuse them repeatedly in your C++ Program. The most common function is the main function that starts a C++ program, which we used previously in Lesson 1. A program may have many functions. Each function has a dedicated purpose, and some action to perform. Functions usually are defined at the top of the program in order as they are used. The main function is the last one because it will call all the proceeding functions. When a function is called in a programming statement it means it is executed. C++ also has many built in functions that you can use, that make C++ programming easier to do, that you will learn later through these lessons. It is now time to add functions to our previous Lesson 1 program. We will make a welcome, enterName, enterAge and printDetails functions. Before proceeding, you may want to save your previous Main.cpp file as Lesson1.cpp for future reference. With Main.cpp closed Right click on Main.cpp and rename Lesson1.cpp


[CppLessons] - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Management
Projects Symbols Files
Workspace
CppLessons
Sources
Main.cpp
Open Main.cpp
Open with
Rename file...
Remove file from project
Format this file (AStyle)
Reparse this file
Build file
Clean file
Options
Properties...
Logs & others
Code::Blocks Search results Cccc Build log Build messages CppCheck/Vera++ CppCheck/Vera++ messages
Welcome to Code::Blocks!
default



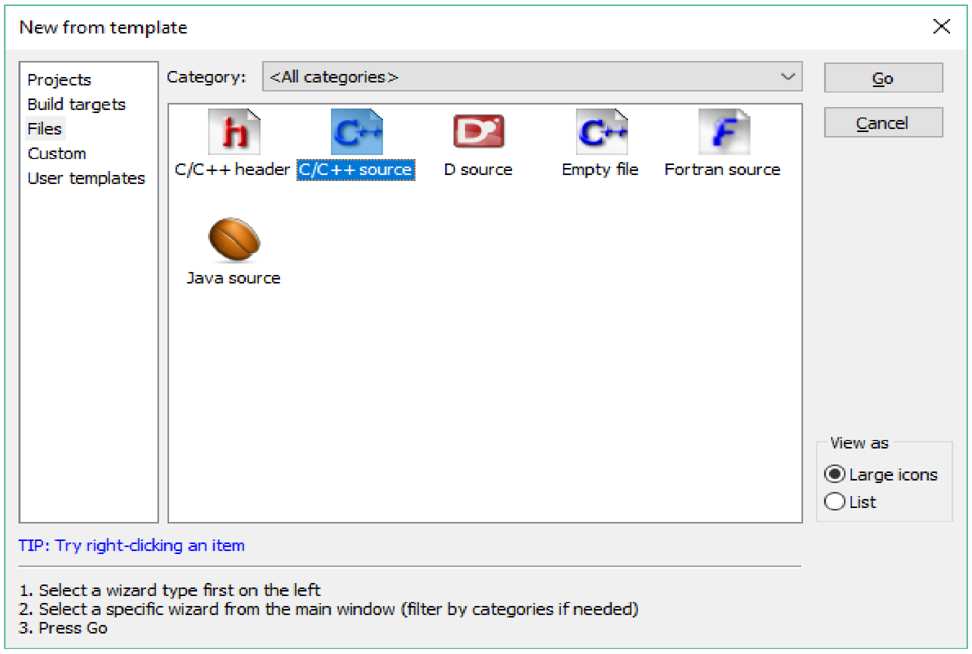
We will now make a new C++ source file called Lesson2.cpp. From the File Menu select New then File.



```
int main()
{
    cout << "Hello world!" << endl;
    cout << "Please type in your name: ";

    string name;
    cin >> name;
    cout << "Nice to meet you " << name << endl;

    cout << "How old are you? ";
    int age = 0;
    cin >> age;
    cout << "You are " << age << " years old" << endl;

    return 0;
}
```

```
-------------- Run: Debug in CppLessons (compiler: GNU GCC Compiler)---------------
Checking for existence: C:\lessons\cpp\CppLessons\bin\Debug\CppLessons.exe
Executing: "C:\Program Files (x86)\CodeBlocks/cb_console_runner.exe" "C:\lessons\cpp\CppLessons\bin\Debug\CppLessons.exe"  (in C:\lessons\cpp\CppLessons\.)
Process terminated with status -1073741510 (0 minute(s), 2 second(s))
```

Select C++ Source File template

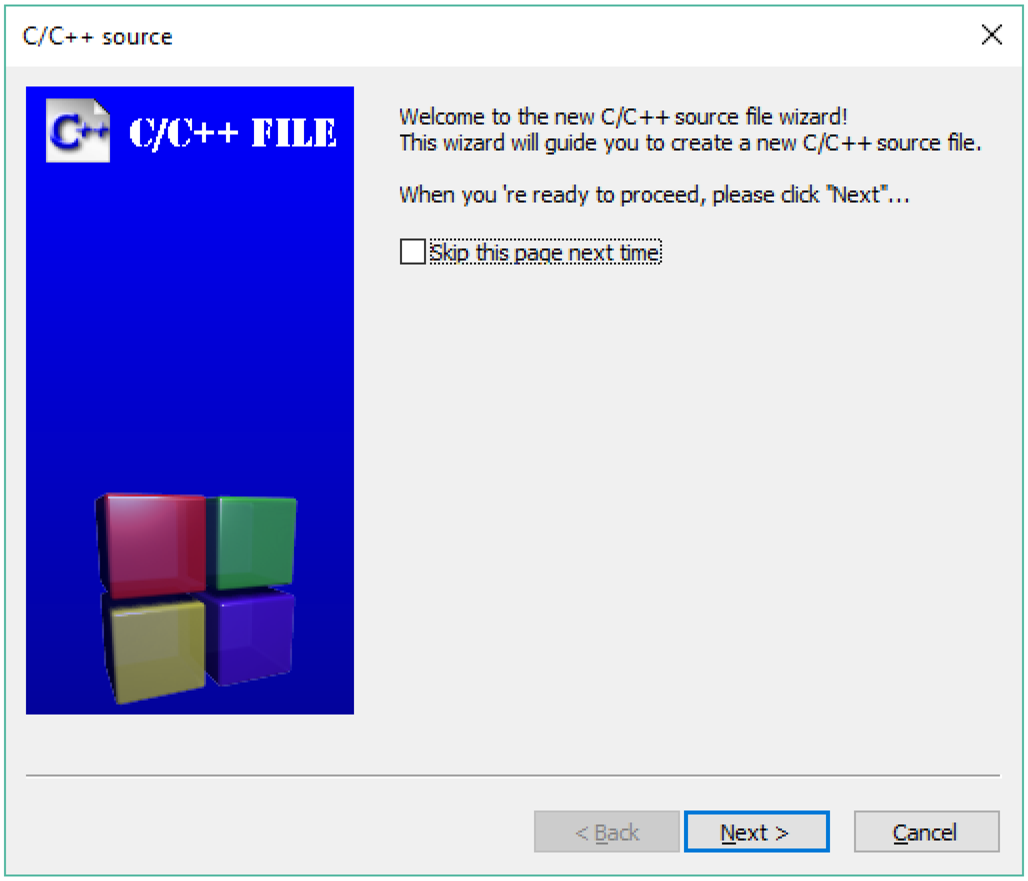


View as: [✓ Large icons] List

TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

Press GO button

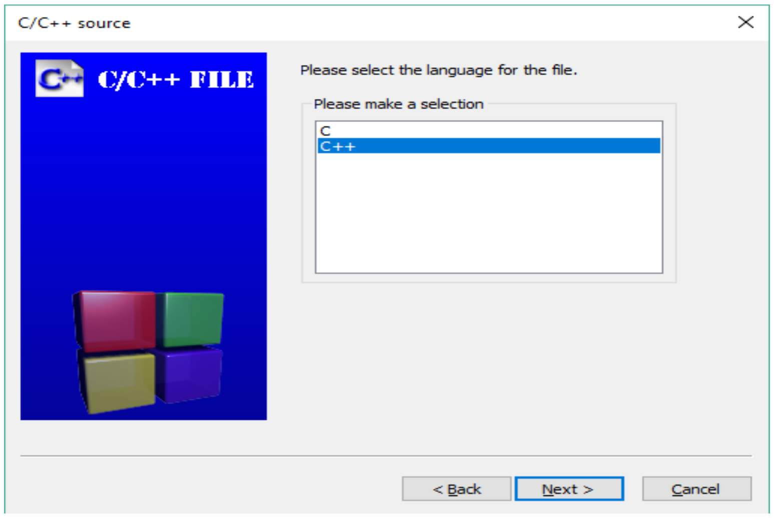

C/C++ source

Welcome to the new C/C++ source file wizard!
This wizard will guide you to create a new C/C++ source file.

When you 're ready to proceed, please click "Next"...

Press Next button

Select C++

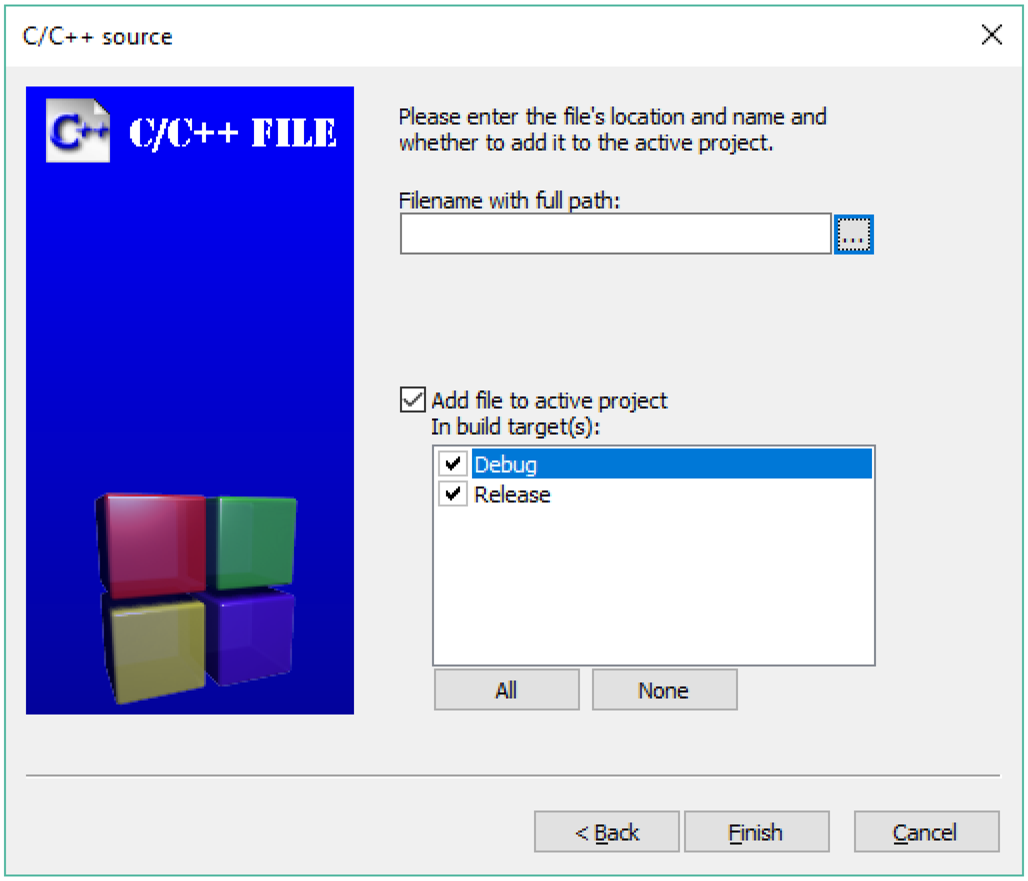

C/C++ source

Please select the language for the file.

Please make a selection

C
C++

Press Next


C/C++ source

Please enter the file's location and name and whether to add it to the active project.

Filename with full path:

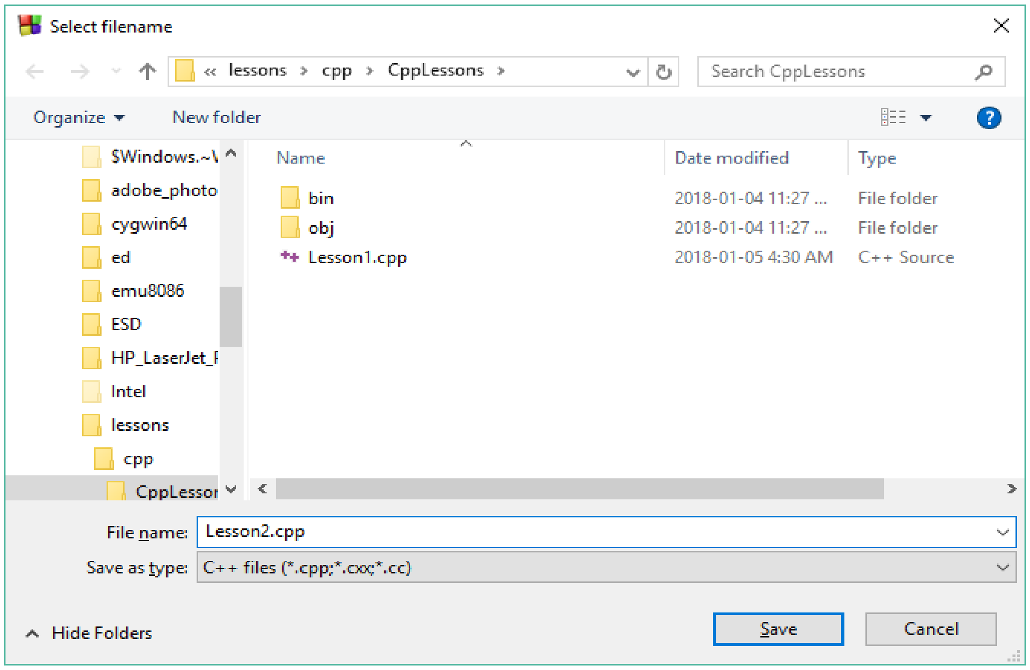
☑ Add file to active project

In build target(s):

☑ Debug

☑ Release

Check Debug and Release Checkbox's, then select file browse button […]
And enter file name Lesson2.cpp


Select filename

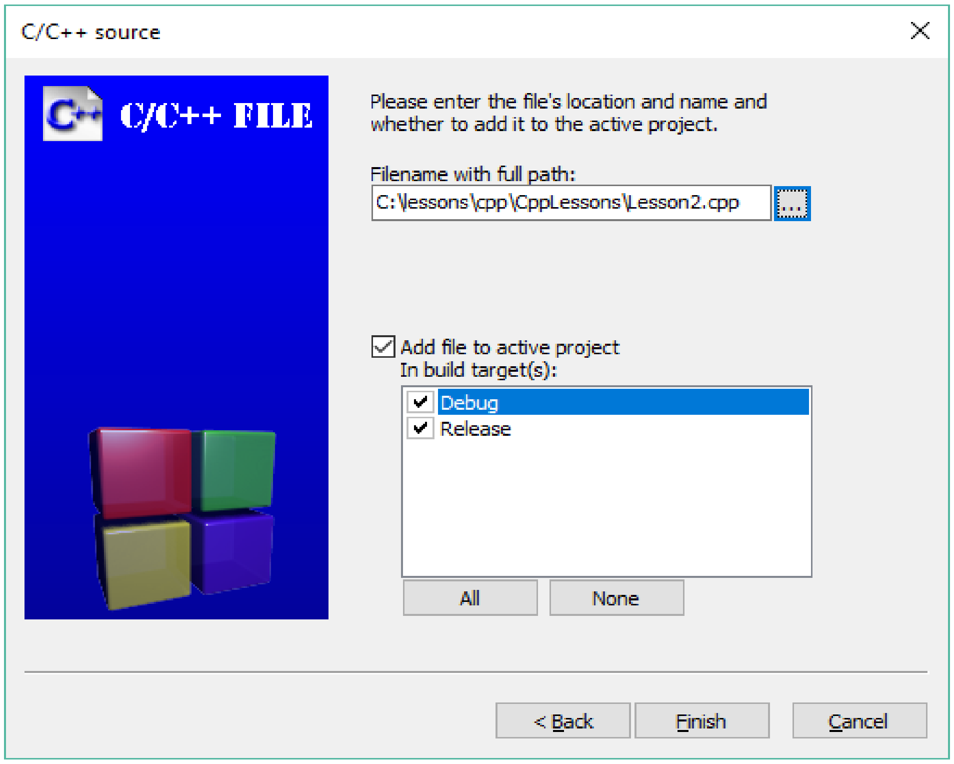
« lessons › cpp › CppLessons ›

Name	Date modified	Type
bin	2018-01-04 11:27 ...	File folder
obj	2018-01-04 11:27 ...	File folder
Lesson1.cpp	2018-01-05 4:30 AM	C++ Source

File name: Lesson2.cpp

Save as type: C++ files (*.cpp;*.cxx;*.cc)

Press Save Button, you should get the following screen


C/C++ source

Please enter the file's location and name and whether to add it to the active project.

Filename with full path: C:\lessons\cpp\CppLessons\Lesson2.cpp

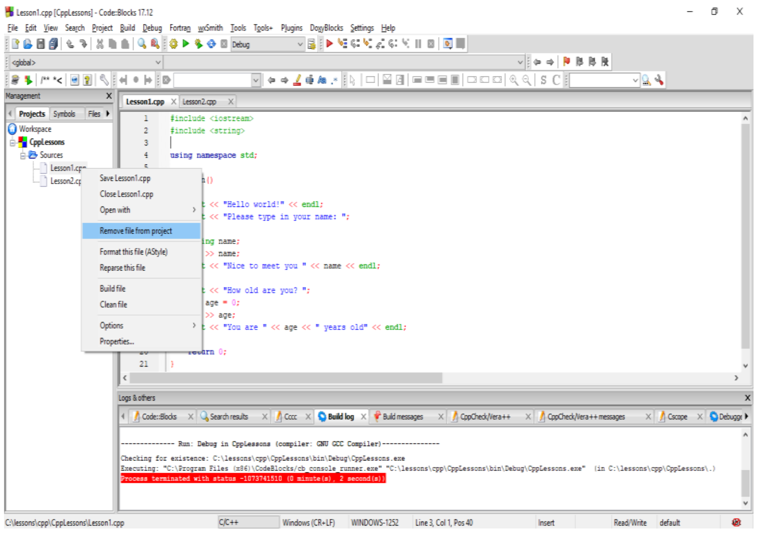
[✓] Add file to active project

In build target(s):

[✓] Debug

[✓] Release

Before pressing Finish button make sure The Debug and Release check boxes are checked. You now need to remove the Main.cpp file or Lesson1.cpp file from the project. A project can only have 1 cpp file with a main function.


Lesson1.cpp [CppLessons] - Code::Blocks 17.12

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

using namespace std;

n()

t << "Hello world!" << endl;
t << "Please type in your name: ";

ing name;
>> name;
t << "Nice to meet you " << name << endl;

t << "How old are you? ";
 age = 0;
>> age;
t << "You are " << age << " years old" << endl;

    return 0;
}
```

```
-------------- Run: Debug in CppLessons (compiler: GNU GCC Compiler)---------------

Checking for existence: C:\lessons\cpp\CppLessons\bin\Debug\CppLessons.exe
Executing: "C:\Program Files (x86)\CodeBlocks/cb_console_runner.exe" "C:\lessons\cpp\CppLessons\bin\Debug\CppLessons.exe"  (in C:\lessons\cpp\CppLessons\.)
Process terminated with status -1073741510 (0 minute(s), 2 second(s))
```

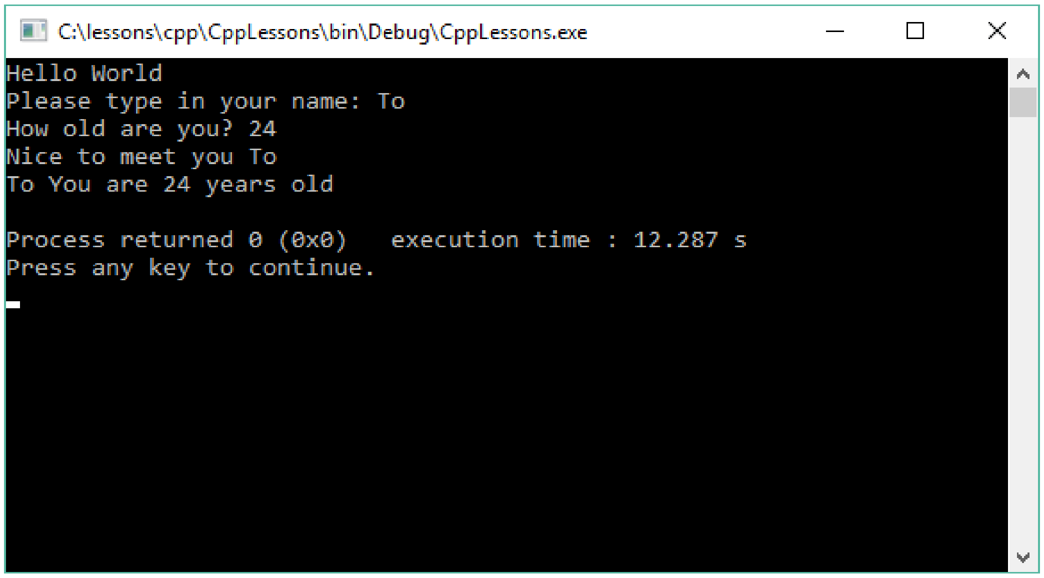
C:\lessons\cpp\CppLessons\Lesson1.cpp | C/C++ | Windows (CR+LF) | WINDOWS-1252 | Line 3, Col 1, Pos 40 | Insert | Read/Write | default

In your Lesson2.cpp file type in the following code.

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

void welcome();
string enterName();
int enterAge();
void printDetails(string name, int age);

 int main()
{
   welcome();
   string name = enterName();
   int age = enterAge();
   printDetails(name, age);
   return 0;
}

void welcome()
{
   cout << "Hello World" << endl;
}

string enterName()
{
   cout << "Please type in your name: ";
   string name;
   cin >> name;
   return name;
}
```

```
int enterAge()
{
   cout << "How old are you? ";
   int age;
   cin >> age;
   return age;
}

void displayDetails(string name, int age)
{
   cout << "Nice to meet you " <<  name << endl;
   cout << name << " You are " << age << " years old" << endl;
}
```

You should now build and run the program. Enter Tom for name and 24 for age, you should get this:



```
Hello World
Please type in your name: To
How old are you? 24
Nice to meet you To
To You are 24 years old

Process returned 0 (0x0)   execution time : 12.287 s
Press any key to continue.
```

Functions make your program more organized and manageable to use. Functions have many different purposes. Function can receive values, return values, receive and return values or receive or return nothing. Function syntax is as follows:

return_datatype function_name (parameter_list)
Parameter list = data_type parameter_name [,data_type parameter_name]

Functions return values using the **return** statement and receive values through the **parameter list**. The **data type** specifies what kind of data is returned or received. In Lesson 1 we were introduced to the int, float, double and string data types.

In C++ before you can use a function you need to declare it. A function declaration is just the function definition header ending in a **semicolon**. A function declaration is also known as a function prototype. Here are the function prototypes for our lesson2 program.

```
void welcome();
string enterName();
int enterAge();
void printDetails(string name, int age);
```

The welcome function just prints “Hello World” and receives no values or returns no value. The void data type specifies no value is returned or received.

```
void welcome()
{
   cout << "Hello World" << endl;
}
```

The enterName() and enterAge() functions both return a value using the return statement. The enterName() function returns a **string** value where as the enterAge function returns an **int** value. Both of these functions have local variables to hold the value obtained from the key board. Local variables are known to the function only and cannot be accessed by other program code. The **enterName** function has the local string variable **name** to hold the name obtained from the keyboard. The **enterAge** function has the local string variable **age** to hold the age obtained from the keyboard. Variables have names that represent what values they are holding.

```
string enterName()
{
   cout << "Please type in your name: ";
   string name;
   cin >> name;
   return name;
}

int enterAge()
{
   cout << "How old are you? ";
   int age;
   cin >> age;
   return age;
}
```

The printDetails function receives a name and age value to print out, but return's no value. The printDetails function receives the name and age through the parameter list. Parameters hold received values and can be used just like a variable inside a function.

```
void printDetails(string name, int age)
{
   cout << "Nice to meet you " <<  name << endl;
   cout << name << " You are " << age << " years old" << endl;
}
```

The **name** and **age** inside the round brackets of the **printDetails** function definition statement are known as **parameters** and contain values to be used by the function. The parameters just store values from the calling function and are not the same variables that are in the calling function. Although the parameter names and values may be same as in the calling function variable names, but they are different memory locations. The main purpose of the parameters is to receive values for the functions.

The main function call's the preceding functions to run them and store the values in its local variables and passes the stored variable values to the functions. Calling a function means to execute the function. The values that are passed to the **called** function from the **calling** function is known as **arguments**.

As stated previously variables inside a function are known as **local variables** and are known to that function only. The **name** and **age** are local variables in the main function but are also used as arguments to the printDetails function.

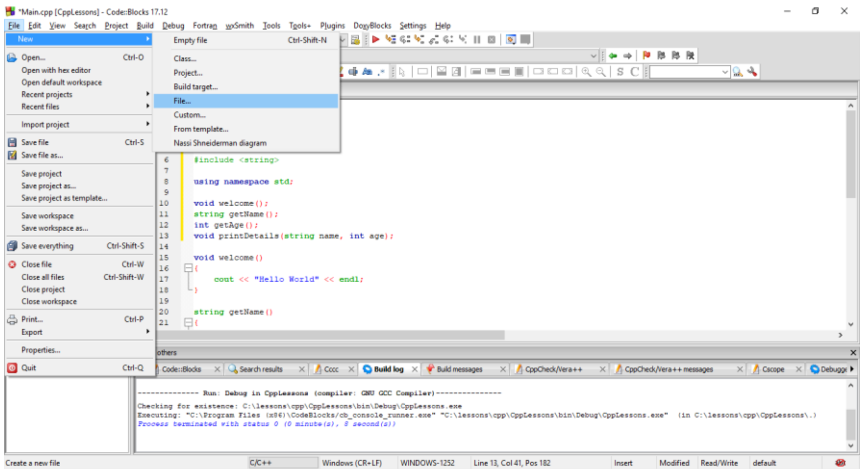
```
int main()
{
   welcome();
   string name = enterName();
   int age = enterAge();
   printDetails(name, age);
   return 0;
}
```

Programming is all about storing values in variables and giving those values to other functions so that they can process the data values.

Function prototypes are usually put into a header file. You should also do the same. Make a file called C++ Header file called Lesson2.h and put in the include statements and function prototypes as follows.

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

void welcome();
string enterName();
int enterAge();
void printDetails(string name, int age);
```

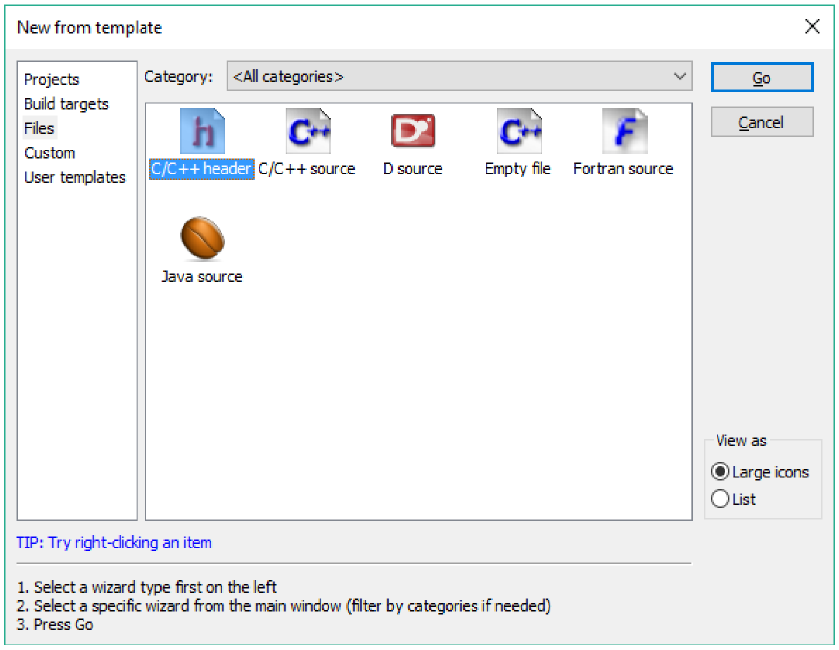
From the File menu select New then File



```
#include <string>

using namespace std;

void welcome();
string getName();
int getAge();
void printDetails(string name, int age);

void welcome()
{
    cout << "Hello World" << endl;
}

string getName()
{
```

```
-------------- Run: Debug in CppLessons (compiler: GNU GCC Compiler)---------------
Checking for existence: C:\lessons\cpp\CppLessons\bin\Debug\CppLessons.exe
Executing: "C:\Program Files (x86)\CodeBlocks/cb_console_runner.exe" "C:\lessons\cpp\CppLessons\bin\Debug\CppLessons.exe"  (in C:\lessons\cpp\CppLessons\.)
Process terminated with status 0 (0 minute(s), 8 second(s))
```

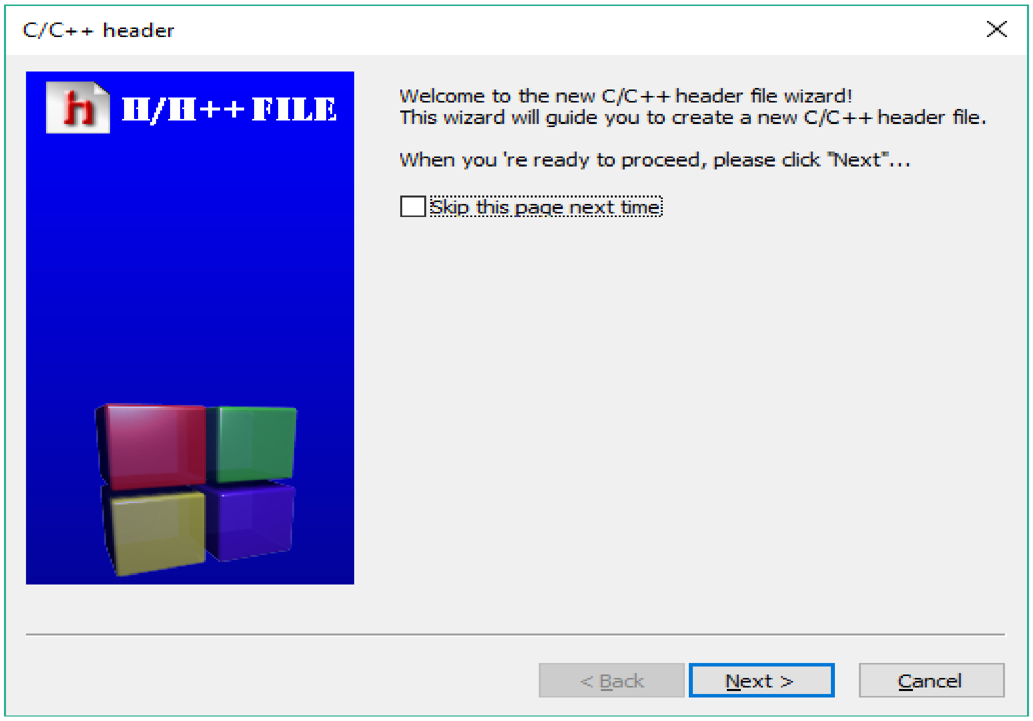
Select the C/C++ Header file type.



TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

Press Go

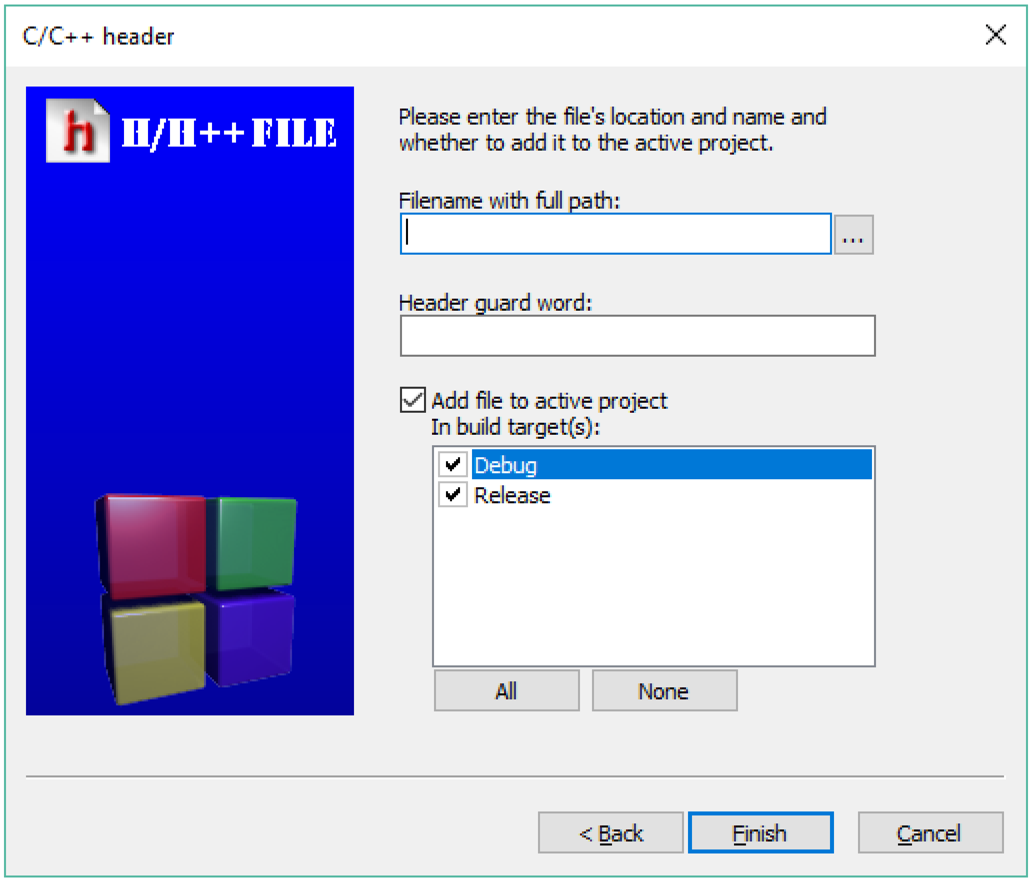

C/C++ header

Welcome to the new C/C++ header file wizard!
This wizard will guide you to create a new C/C++ header file.

When you 're ready to proceed, please click "Next"...

☐ Skip this page next time

Press Next


C/C++ header

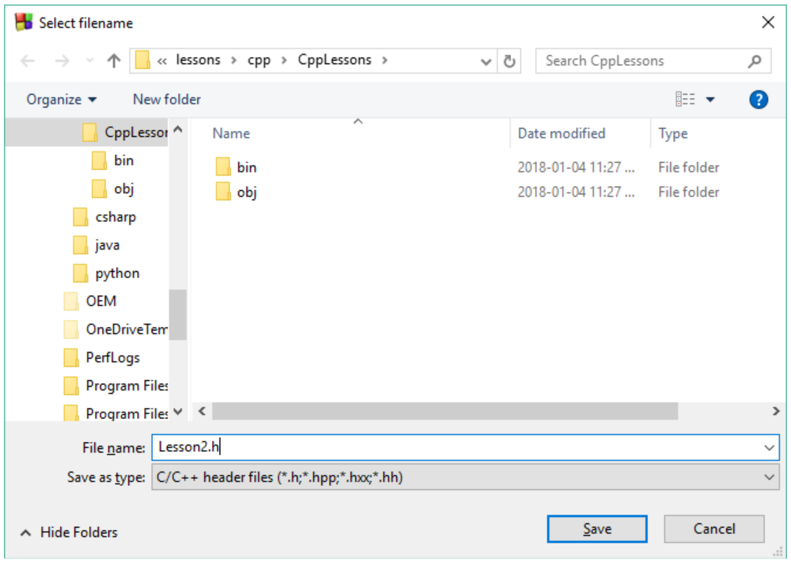
Please enter the file's location and name and whether to add it to the active project.

Filename with full path:

Header guard word:

☑ Add file to active project
In build target(s):
☑ Debug
☑ Release

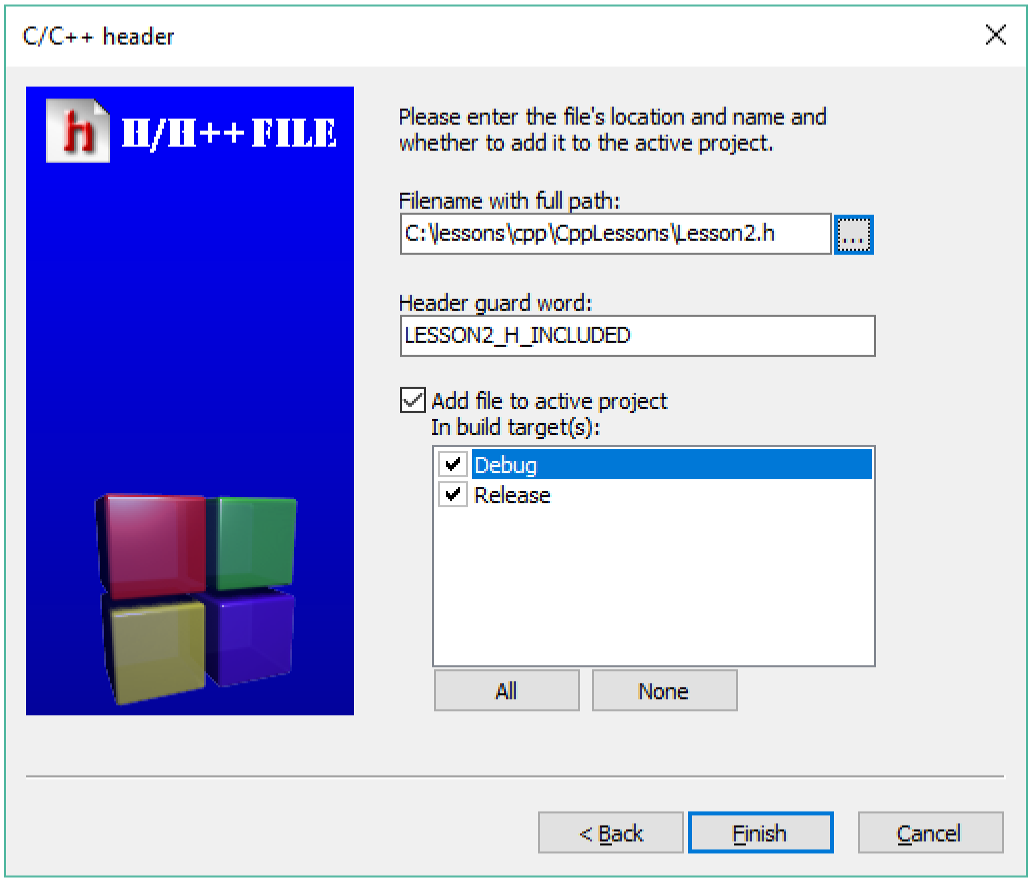
Make sure the Debug and Release check boxes are checked before proceeding.
Select the filename browse button … then type in Lesson2.h



Name	Date modified	Type
bin	2018-01-04 11:27 ...	File folder
obj	2018-01-04 11:27 ...	File folder

File name: Lesson2.h
Save as type: C/C++ header files (*.h;*.hpp;*.hxx;*.hh)

Press Next



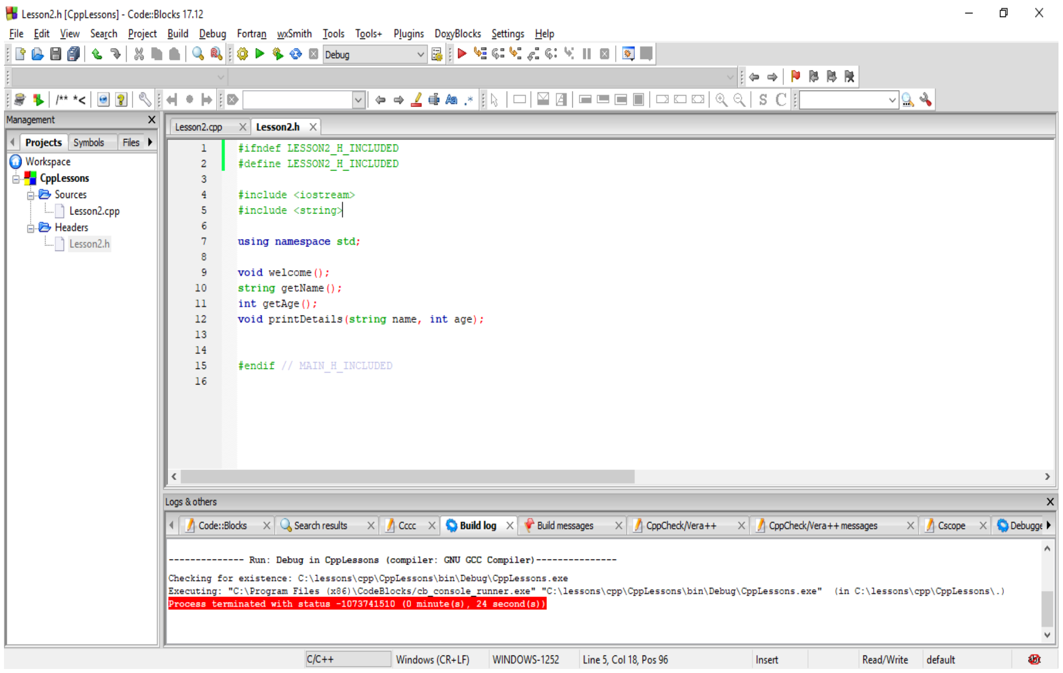
Please enter the file's location and name and whether to add it to the active project.

Filename with full path: C:\lessons\cpp\CppLessons\Lesson2.h

Header guard word: LESSON2_H_INCLUDED

☑ Add file to active project
In build target(s):
✓ Debug
✓ Release

Make sure Debug and Release are checked then Press Finish
Your Lesson2.h should look like this after typing in the function prototypes.



```
#ifndef LESSON2_H_INCLUDED
#define LESSON2_H_INCLUDED

#include <iostream>
#include <string>

using namespace std;

void welcome();
string getName();
int getAge();
void printDetails(string name, int age);


#endif // MAIN_H_INCLUDED
```

```
-------------- Run: Debug in CppLessons (compiler: GNU GCC Compiler)---------------

Checking for existence: C:\lessons\cpp\CppLessons\bin\Debug\CppLessons.exe
Executing: "C:\Program Files (x86)\CodeBlocks/cb_console_runner.exe" "C:\lessons\cpp\CppLessons\bin\Debug\CppLessons.exe"  (in C:\lessons\cpp\CppLessons\.)
Process terminated with status -1073741510 (0 minute(s), 24 second(s))
```

Lesson2.h [CppLessons] - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
Management
Projects Symbols Files
Workspace
CppLessons
Sources
Lesson2.cpp
Headers
Lesson2.h
Logs & others
Code::Blocks Search results Cccc Build log Build messages CppCheck/Vera++ CppCheck/Vera++ messages Cscope Debugge
C/C++ Windows (CR+LF) WINDOWS-1252 Line 5, Col 18, Pos 96 Insert Read/Write default

#ifndef LESSON2_H_INCLUDED
#define LESSON2_H_INCLUDED

Are known as guards and allow the .h file only to be read once. Without the guards the .h file may be read many times and resulting in duplicate function declaration error messages.

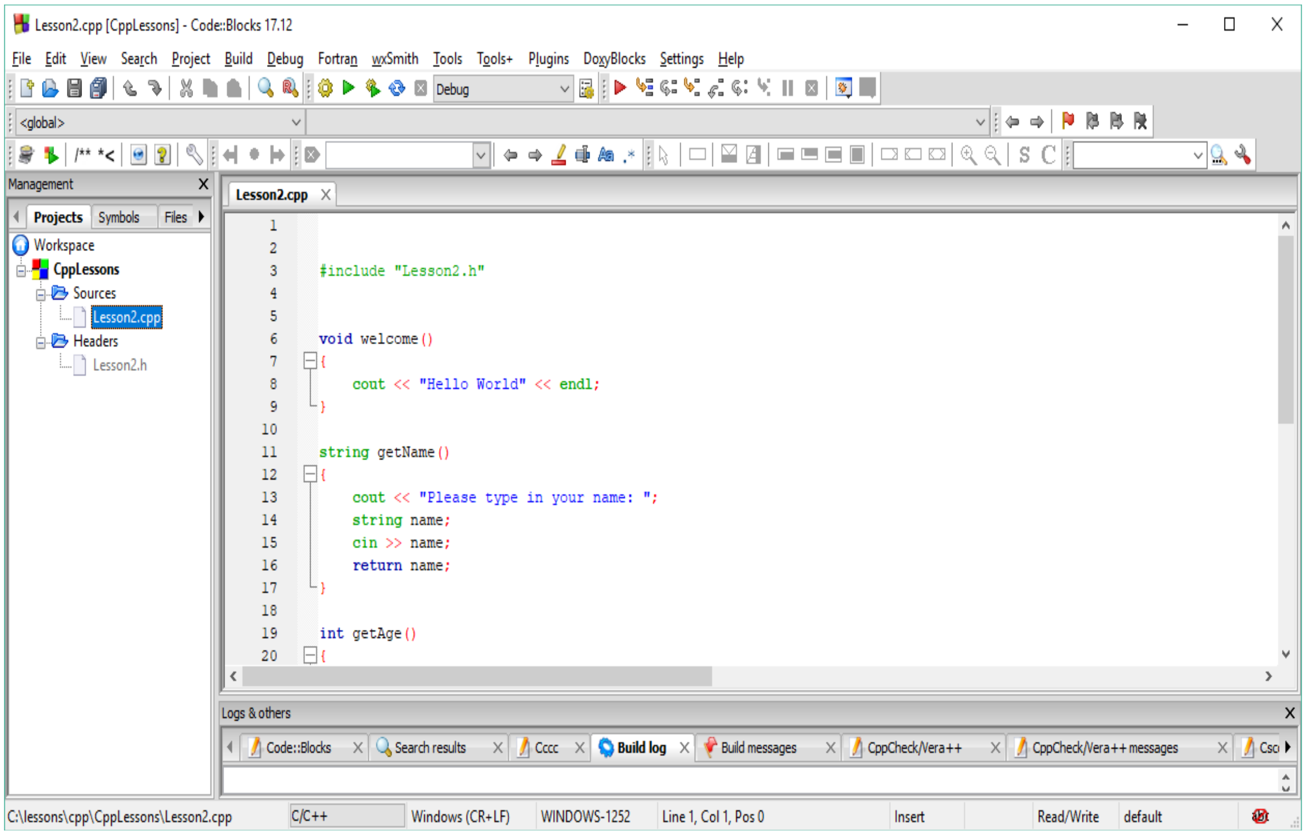
The guard ends with

#endif

You now need to remove the include statements and function prototypes on the top of the Lesson2.cpp file since they are no longer needed. You need to include the Lesson2.h file at the top of your main.cpp instead like this:

#include "Lesson2.h"

So that the cpp file can read the function prototypes from the Lesson2.h file.



```
#include "Lesson2.h"


void welcome()
{
    cout << "Hello World" << endl;
}

string getName()
{
    cout << "Please type in your name: ";
    string name;
    cin >> name;
    return name;
}

int getAge()
{
```

C:\lessons\cpp\CppLessons\Lesson2.cpp   C/C++   Windows (CR+LF)   WINDOWS-1252   Line 1, Col 1, Pos 0   Insert   Read/Write   default

It's now time to comment your program. All programs need to be commented so that the user knows what the program is about. Just by reading the comments in somebody will know exactly what the program is supposed to do. We have two types of comments in C++. Multi-line comments that are usually at the start of a program or a function. They start with /* and end with a */ and can span multiple lines like this:

/*
Program to read a name and age from a user and
print the details to the screen
***/**

Other comments are for one line only and explain what the current or proceeding program statement it is to do. The one-line comment starts with a // like this:

// function to read a name from the key board are return the value

We now comment the program. Please add all these comments to your program.

```
/*
        Program to read a name and age from a user and print
        the details on the screen
*/

#include "Lesson2.h"

// function to print welcome message
void welcome()
{
   cout << "Hello World" << endl;
}

// function to obtain a name from the keyboard
string enterName()
{
   cout << "Please type in your name: ";
   string name;
   cin >> name;
   return name;
}

// function to obtain an age from keyboard
int enterAge()
{
   cout << "How old are you? ";
   int age;
   cin >> age;
   return age;
}

// function to print name and age on screen
void printDetails(string name, int age)
{
   cout << "Nice to meet you " <<  name << endl;
   cout << name << " You are " << age << " years old" << endl;
}
```

```
int main()
{
   welcome();   // welcome user
   string name = enterName();  // obtain a name
   int age = enterAge();  // obtain an age
   printDetails(name, age); // print out name and age
   return 0;
}
```

C++ Lesson 2 Homework

Make a C++ program file called homework2.cpp that has functions to ask someone what their profession title is, annual salary is and to print out the information. Print out a message like this: "I am a Manager and I make $100,000 dollars per year! ". Use a string for the profession title variable. Use a double data type for salary variable. You should have functions welcome, enterProfession, enterSalary and printDetails(). Call all the functions from the main function. Put your homework2.cpp file in a C++ project called Homework2.

LESSON 3   CLASSES

Classes represent another level in program organization. They represent programming units that contain variables to store values and functions to do operations on these variables. This concept is known as **Object Oriented Programming** and is a very powerful concept. It allows these programming units to be used over again in other programs. The main benefit of a class is to store values and do operations on them transparent from the user of the class. It is very convenient for the programmers to use classes. They are like building blocks that are used to create many sophisticated programs with little effort.

A class is first defined in a header file and implemented in a cpp file. A class definition is similar to a function prototype but also contains variable and constant definitions. A class starts with the keyword **class** and the class name. Following the class definition are the constants and variables. Constants are values once initialized never change and variables are used to store values that can be changed. Constants and variables are usually private meaning they can only used by the class internally. Private variables are a mechanism known as **encapsulation** which means some one is using the class but does not know anything about the internal variables, they are hidden from the user. Following the variables are the constructor's and functions. Constructors are used to initialize the variables defined in the class, functions are used to access the variables and to do operations on the variables. These operations may include addition, incrementing etc. Functions define in a class are also known as **methods**. To avoid confusion, we will call functions declared in a class to be called methods and functions not belonging to a class still to be called functions. Constructors and methods are usually public meaning this can be used externally by other methods. Some methods may also be private, meaning that can only be used internally, only by methods belonging to the class.   The class definition ends with a semicolon. Do not forget the semicolon!

The class syntax as follows:

```
class class_name
        {
        private:
            constant declarations
            variable declarations
        public:
            constructor declarations
            method declarations
        };
```

As mentioned previously a class is like a user defined data type. But the class is much more powerful because it can represent a value and have methods to do things (operations) on its variables. A class specifies what variables and methods an object is going to have, just like an architect designs plans for a house on paper and defines how many rooms the house will have. When the house is built from the plans it becomes a house object that some one could live in. When your program runs memory is allocated for the variables defined in the class that was specified in your program and the values live in the memory locations. When the program runs the memory locations holding variables defined in the class are now called **objects**. We will now define a Person class that has variables to store a name and age and methods to do operations on them. These operations include initializing retrieval, assignment and output. Usually a class definition file is put in a .h header file and the class implementation file is put into .cpp source file.

Make a new C++ header file called Person.h, and type the following code into it. Make sure Debug and Release are checked before you press the Finish button.

```
#ifndef PERSON_H_INCLUDED
#define PERSON_H_INCLUDED

/*
Person Class to store a person's name and age
*/

#include <iostream>
```

```
#include <sstream>
#include <string>
using namespace std;

// define a class Person
class Person
{
private:
   string name;
   int age;
public:

   // initialize a default Person
   Person();

   // initialize Person
   Person(string name, int  age);

  // return  name
   string getName();

   // assign name
   void setName(string name);

   // return age
   int getAge();

   // assign age
   void setAge(int age);

   // return person info as a string
   string toString();
};

#endif // PERSON_H_INCLUDED
```

The Person class definition starts with the **class** key word and class name Person.

```
class Person{
```

Our Person class has 2 private variables to store person name and age.

```
private:

    string name;
    int age;
```

We make the variables **private** because we want them to be only access by our class methods, nobody else.

The following constructor and methods are public because they will be used **externally** by other methods. Next, we declare the constructor's whose only purpose in life is to initialize the variables in the class. We have many types of constructors. Default, initializing and copy constructors. Default contractors would initialize an object to a default value. An initializing constructor would initialize an object to some know values passed to it. Initializing constructors are also known as parameter constructors because they have parameters to receive values. A copy constructor would copy values from an existing object. Constructors are considered special methods. Methods that have the same name, but different parameter lists are known as **overloaded** methods. This is the default constructor declaration; no values are passed to it:

```
Person();
```

This is the initializing constructor, it gets 2 parameter values to initialize the object with, a name and an age:

```
Person(string name, int  age);
```

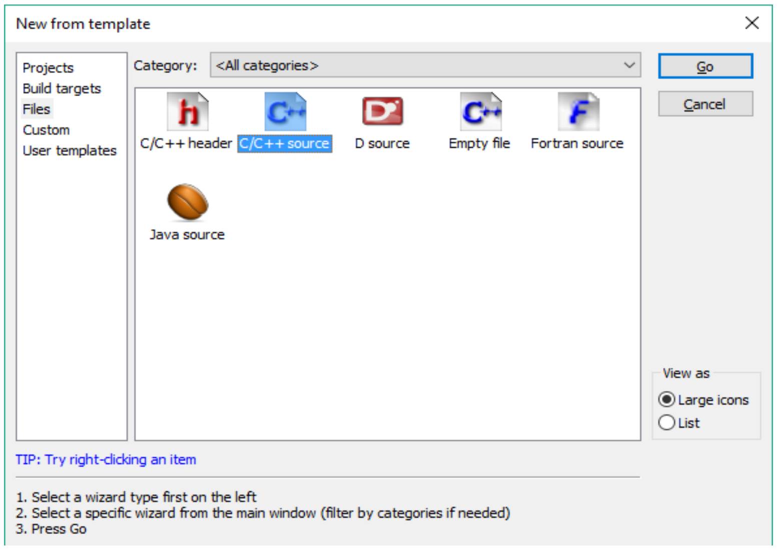
We next declare the get and set methods. The **get** methods are known as **getters** because the get and return values where the **set** methods are known as **setters** because they assign values to the variables declared in the class, **setters** are also known as **mutators**, meaning mutate values (change values).

```
string getName();
void setName(string name);
int getAge();
void setAge(int age);
```

Lastly, we have the toString method that return a string info about the class. All classes should have toString() method.

```
string toString();
```

The class implementation file is put into a cpp file. Make a new C++ source file called Person.cpp. Make sure Debug and Release are checked before you press the Finish button or else they will not be compiled into the project.



TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

Type the following into the Person.cpp file

```
/*
Person Class to store a person's name and age
*/
```

```
#include "Person.h"

// default Person
Person::Person()
{
   this->name = "";
   this->age = 0;
}

// initialize Person
Person::Person(string name, int  age)
{
   this->name = name;
   this->age = age;
}

// return name
string Person::getName()
{
   return this->name;
}

// assign name
void Person::setName(string name)
{
   this->name=name;
}

// return age
int Person::getAge()
{
   return this->age;
}
```

```
// assign age
void Person::setAge(int age)
{
   this->age = age;
}
// return person info as a string
string Person::toString()
{
   ostringstream sout;
   sout << "Nice to meet you " << this->name << endl;
   sout << this->name << " You are " <<  this->age <<  " years old" << endl;
   return sout.str();
}
```

The class implementation file starts with

```
#include "Person.h"
```

The Person.h file includes the Person class definition that the Person.cpp source file needs to know about.

Each method name in the implementation file starts with Person:: the :: is known as a **resolution operator** which tells the compiler which method belongs to which class. The class implementation file uses another keyword called **this** that indicates that a variable or methods belong to the defining class.

A constructor is used to initialize a class and has the same name as the class. The following is the default constructor that initializes our Person class to some default values.

```
// default Person
   Person::Person() {
      this->name = "";
      this->age = 0;
   }
```

The programming statements inside the default constructor assign values to the variables name and age having default value empty string and 0 age.

```
    this->name = "";
    this->age = 0;
```

The initializing constructor initializes the Person class to the name and age values passed to it.

```
// initialize Person
  Person::Person(string name, int  age)
  {
    this->name = name;
    this->age = age;
  }
```

The programming statements inside the initializing constructor assign values to the variables name and age from the parameters name and age.

```
    this->name = name
    this->age = age
```

The keyword **this** specifies which variables belongs to the Person class. The parameter name and age just store values to be assigned to the Person class variables and are not the same ones in the Person class. The **this** keyword differentiates the variables defined in the class definition and the parameters. Without using the this keyword the variables defined in the class would never be initialized. Alternatively you could use different names for the parameter then the this keyword is not necessary. It is probably better to use the this keyword since you will always know which variables were defined in the class definition.

The get methods return the values of the variables defined in the class.
Get methods are also known as getters.

```
// return name
string Person::getName()
{
   return this->name;
}

// return age
int Person::getAge()
{
   return this->age;
}
```

The **this** keyword indicate the variables that were defined in the class.

The **set** methods assign values the variables defined in the class. Set methods are also known as **setters** or **mutators**.

```
// assign name
void Person::setName(string name)
{
   this->name=name;
}

// assign age
void Person::setAge(int age){
   this->age = age;
}
```

We use the **this** keyword to distinguish between a variable defined in the class and the parameter since they both have the same name. The other alterative is to use a different name for the parameter which most people do. Using the **this** keyword is a more modern approach but may be too professional for most people.

All classes should have a **toString()** method so that it can easily return the class info as a string message.

```
string Person::toString(){
   ostringstream sout;
   sout << "Nice to meet you " << this->name << endl;
   sout << this->name << " You are " <<  this->age <<  " years old" << endl;
   return sout.str();
}
```

Inside we use the **ostringstream** class to store our info message. We instantiate a ostringtream object by declaring the sout variable having the data type ostringsteam. We use the sout variable just as we used the cout variable. This is a handy situation to be in. The alterative is to use a string variable. The problem with a string variable is that you cannot join string and numeric values. In order to use a string variable, we would have to convert all numeric values to string values. The ostringsteam class solves all our problems and can easily joining string and numeric values together. The sout variable returns a string by calling the **str()** method of the ostringstream class.

Usually a class definition must not contain any input or output statements. A class must be a reusable program unit not dependent on any input or output print statements. The purpose of the class is to contain information that can be easily accessed. Therefore, our main function must provide all the input and output print statements. We will use the input and output functions from our previous program. Make a new C++ source file called Lesson3.cpp and type in the following code.

```
/*
 Program to read a name and age from a user and print
 the details on the screen
*/

#include "Lesson3.h"
#include "Person.h"
```

```
// function to print welcome message
void welcome()
{
   cout << "Hello World" << endl;
}

// function to obtain a name from the keyboard
String enterName()
{
   cout << "Please type in your name: ";
   string name;
   cin >> name;
   return name;
}

// function to obtain an age from keyboard
int enterAge()
{
   cout << "How old are you? ";
   int age;
   cin >> age;
   return age;
}

int main()
{
   welcome();   // welcome user
   string name = enterName();    // obtain a name
   int age = enterAge();    // obtain an age
   Person p(name, age);   // make person object
   cout << p.toString() << endl;   // print out name and age
   return 0;
}
```

Make a new .h header file called Lesson3.h and copy in the contents of Lesson2.h. There is not much change needed to be made from the original Lesson2.cpp file. Build and run the program, type Tom for name and 24 for age. You will get the following output.

```
Hello World
Please type in your name: Tom
How old are you? 24
Nice to meet you Tom
Tom You are 24 years old
```

At the top of the Lesson3.cpp file We must include our person class definition or else the compiler will not be able to know about the Person class.

#include "Person.h"

Much of them main function is not changed and can be used as is.

We create the Person class with the following statement:

Person p(name, age);

This calls the **Person** constructor of the person class to create the person object and initialized with the values name and age. The mechanism that allocates memory in the computer for the variables and method code defined in the class, is known as **instantiation**. When a class is instantiated in computer memory it is known as an **object**. When a class is written in a program then it is still known as a class not an object. Objects are made from class definitions. We can make many objects from the same class definition. Just like a builder can make many houses from the same drawing.

The **cout** statement calls the **toString()** method to print out the Person info.

cout << p.toString() << endl;

You can define a default Person object like this:

```
Person p;
```

It is just like declaring a variable like x

```
int x;
```

Except it is a Person variable p the variable p represents the Person object stored in computer memory created from the class person. Always remember **objects** are made from **class definitions**. Objects represent memory for the variables declared in the class definition when the object is instantiated (created) from the class definition.

Note the () are not included when instantiating a default person object. If you include the round brackets then the compiler thinks you a declaring a person default constructor not instantiating a default person object.

Things to do:

Declare and instantiate a default Person class called p2. Use the getters to obtain values from Person p and the setters to assign values to Person p2. Then print out the p2 Person class info using cout and the toString method. .Do this in your Lesson3.cpp file.

LESSON3 HOMEWORK Question 1

Make an Profession class that stores an profession's job title and salary. Use a string for title and a double data type for salary. Make a default constructor and a initializing constructor that receives a profession title and a salary. Make getters and setters for title and salary. Make a toString() method that prints out the profession details like this:

```
I am a Manager,
 I make $100,000 dollars
```

In your main function function instantiate a Profession object with a title and a salary. Use the enterTitle() and enterSalary() function from homework 2 to get a title and salary from the keyboard.

 Also include a welcome () function that prints out what the program is suppose to do. From the Profession object print out the title and salary using the getters. Next change the profession title and salary using the setters and then print out the profession object again.

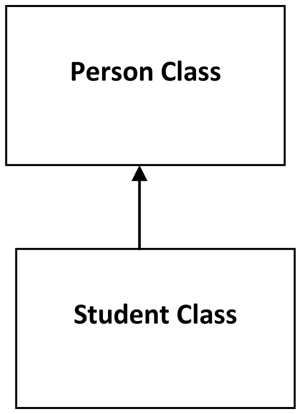
You can put your main function in a file called Homework3.cpp. Make files Profession.h and Profession.cpp for the Profession class.

INHERITANCE

The beauty of classes is that they can be extended to increase their functionality. We can make a Student class that uses the public variables and methods from the Person class. This is known as **inheritance**. The Student class can only access the public variables and methods from the Person class. We have additional access modifiers public, protected, and private. The default is private.

access modifier	description	example
Public	Access by anybody	public int age;
Protected	Access by derived class	protected int age;
Private	Access by its own class only	private int age;

A Student class will have an additional variable called **idnum** that will represent a string student id number. Using inheritance, the student class will be able to use the public variables and methods of the Person class. The Person class is known as the **super** or **base** class and the Student class is known as the **derived** class.



Inheritance is a one way street. The Person class knows nothing about the Student class where as the Student class knows all about the Person class.

Create a new C++ header file called Student.h, make sure Debug and Release are checked before pressing the finish button.

Create a class called Student that inherits from the Person class using the following statement.

```
// define a class Student that inherits the Person class
class Student: public Person {
```

The colon **:** specifier **and public keyword** is used to define the inheritance relationship. This means the Student class can use the public variables and methods from the Person class. We now need to define a student id number variable for the Student class.

```
// student id number
private:
    string idnum;
```

We define a default constructor that will initialize the student name, age and idnum to default values.

```
// default Student
Student();
```

We now define a constructor that will initialize the student name, age and idnum.

```
// initialize Student
Student(string name, int  age, String idnum);
```

The student ID getter and setters are as follows:

```
        // return student id
        String getIDnum();

        // assign student id
        void setIDnum(string idnum);
```

Lastly declare a toString method

```
  // return student info string
  string toString();
```

Here is the complete Student class definition:

```
#ifndef STUDENT_H_INCLUDED
#define STUDENT_H_INCLUDED

#include "Person.h"

class Student: public Person {

// student id number
private:
    string idnum;

  public:

   // initialize Student
  Student(string name, int  age, string idnum);

  // return idnum
  string getID();

  // assign idnum
  void setID(string idnum);

  // return student info string
  string toString();
};

#endif // STUDENT_H_INCLUDED
```

Now make a C++ Source file called Student.cpp. Make sure Debug and Release are checked before pressing the finish button. On the top of Student.cpp you will need to include the Student.h file.

```
#include "Student.h"
```

The Student default constructor must also call the Person default constructor using the initializing operator : so that the Person object also gets initialized. The idnum of the Student class is initialized to the default value of 0.

```
Student::Student() : Person()
   {
      this->idnum = 0;
   }
```

The Student initializing constructor must also call the Person initializing constructor using the initializing operator : so that the Person object also gets initialized. The name and age parameters of the Student constructor are passed to the Person constructor so that the Person object can be initialized also. The idnum of the Student class is initialized to the value inside the parameter idnum.

```
Student::Student(string name, int  age, string idnum)
   : Person(name, age)
   {
      this->idnum = idnum;
   }
```

The colon : initializing operator calls the base constructor of the Person class to create a Person object and transfer the name and age values from the parameters name and age of the Student constructor. The idnum will be initialized in the Student constructor.

Here are the Student setters and getters:

```
// return student id
String getIDnum()
{
   return this->idnum;
}
```

```
// assign student id
void setIDnum(string idnum)
{
   this->idnum=idnum;
}
```

The last thing you need to make the **toString()** method. By using the super class name and resolution operator you can call methods directly from the super Person class inside the Student derived class.

Here is the Student **toString()** method calling the Person super class toString method to get all the common info.

```
// return student info as a string
   string Student::toString()
      {
      string s  = Person::ToString();
      s += " Your student id number is " +  this->idnum;
      return s;
     }
```

Here is the complete Student.cpp file:

```
#include "Student.h"

  Student::Student(string name, int  age, string idnum)
      : Person(name, age)
      {
          this->idnum = idnum;
      }

      // return idnum
   string Student::getID()
   {
      return idnum;
   }
```

```
// assign idnum
void Student::setID(string idnum)
{
   this->idnum = idnum;
}

// return student info string
string Student::toString()
{

   string sout = Person::toString();
   sout += " Your student id number is " +  this->idnum;
   return sout;
}
```

Testing Student class:

Before testing the Student class we need to make a function to get a Student Id from the keyboard.

```
// function to obtain a student ID from the keyboard
String enterStudentID()
{
   cout << "Please type in your student ID: ";
   string studentID;
   cin >> studentID;
   return studentID;
}
```

Now we can get a student id from the keyboard using our enterStudentID() function.

```
String idnum = enterStudentID();
```

Next make a default Student

```
Student s;
```

Then make a initialized student

```
Student  s2(name, age, idnum);
```

Then print out the student

```
cout << s2.toString() << endl;
```

You should get something like this:

```
Please type in your name: Sue
How old are you? 26
Please type in your ID: S1234
Nice to meet you Sue
Sue You are 26 years old
Your student id number is S1234
```

Things to do:

Declare and instantiate a default Student class called s2. Use the getters to obtain values from Student s and the setters to assign values to Student s2. Then print out the s2 Student info using cout and the toString method. Do this in your Lesson3.cpp file.

LESSON3 HOMEWORK Question 2

Make a **Department** class that is derived from your **Profession** class that stores a department name. A department name is the department that the professional works in like the sales department. Make a default constructor and an initializing constructor that receives a profession title, salary and a department name. Make getters and setters for the Department class. Make a **toString()** method that prints out the department info like:

```
I am a Manager,
I make $100,000 dollars
and I work in the Accounting department.
```

In your main function instantiate a Department object. In file Homework3.cpp make an additional function called **enterDepartmentName()** that gets a department name from the keyboard. Update the welcome function that describes what the program is supposed to do.

From the Department object print out the department and title and salary using the getter. Next change the department name using the setters and then print out the department object again.

Make files Department.h Department.cpp and update file Homework3.cpp.

LESSON 4 OPERATORS

Operators

Operators do operations on variables like addition + , subtraction – and comparisons > etc. We now present all the C++ operators with examples. Make a new C++ file called Lesson4.cpp. In your Lesson4.cpp file in the main function, type out the examples and use **cout** statements to print out the results.

Arithmetic Operators

Arithmetic operators are used to do arithmetic operations on numbers like addition and subtraction. You can type in the operation right inside the cout statement just like this

```
cout << (3+2) << endl;
```

Alternatively, you can use variables instead like this:

```
int x = 3;
int y = 2;
cout << x <<  " + " <<  y << " = " << (x + y) << endl;
```

Operator	Description	Example	Result
+	Add two operands or unary plus	3 +2	5
-	Subtract right operand from the left or unary minus	3 - 2 -2	1 -2
*	Multiply two operands	3 * 2	6
/	Divide left operand by the right one	5 / 2	2
%	Modulus - remainder of the division of left operand by the right	5 % 2	3

Comparison Operators

Comparison operators are used to compare values. It either returns true or false according to the condition. True and false variables and values are known as **bool**. True has the value 1 and false has the value 0.

bool b = false;
cout << b << endl; // would print out 0 (false)

You can print out true and false rather than 1 and 0 by using:

cout << boolalpha;
cout << b << endl; // would print out false rather than 0

You can type in the operation right inside the cout statement just like this:
cout << (5 > 3) << endl; // prints out 1 (true)

Alternatively, you can use variables instead like this:
x = 3
y = 2;
cout << x << " > " << y << " = " << (x > y) << endl; // prints out 1 (true)

Operator	**Description**	**Example**	**Result**
>	Greater than - true if left operand is greater than the right	5 > 3	True
<	Less than - true if left operand is less than the right	3 < 5	true
==	Equal to - true if both operands are equal	5 == 5	True
!=	Not equal to - true if operands are not equal	5!= 5	True
>=	Greater than or equal to - true if left operand is greater than or equal to the right	5 >= 3	True
<=	Less than or equal to - true if left operand is less than or equal to the right	5 <= 3	True

Logical Operators

Logical operators are the && (**and**), || (**or**) , ! (***no*****t**) operators. Logical operators form conditions. A Conditions joins two boolean values together using a logical operator. Boolean values are either true or false. The condition is then evaluated as true or false

$$\underbrace{\overset{\textit{true}}{\textit{boolean\_value}}\ \overset{\textit{\&\&}}{\textit{logical\_operator}}\ \overset{\textit{false}}{\textit{boolean\_value}}}_{\text{condition}}$$

You can type in the condition right inside the cout statement just like this:

```
    cout << (true && true) << endl; // prints  1 (true)
```

Alternatively, you can use variables instead like this:

```
    bool bx = true;
    bool by = false;

cout << bx <<  " && " <<  by << " = " << (bx && by) << endl; // prints out  1 (true)
```

use this to print out **true** or **false**

```
cout << boolalpha; // true
```

Operator	Description	Example	Result
&&	true if both the operands are true	true && true	true
||	true if either of the operands is true	true || false	true
!	true if operand is false false if operand is true	! false ! true	true false

Compound conditions

Compound conditions combine **Logical operators** and **comparison operators** to together.

You can type in the operation right inside the cout statement just like this:

```
    cout << (5 > 3 && 3 < 7) << endl;  // prints 0 (false)
    cout << (5 > 3 || 3 < 7) << endl;  // prints 1 (true)
```

Alternatively, you can use variables instead like this:

```
x = 3;
y = 2;

cout << (x== y || x <= y) << endl;  // prints 0 (false)
```

todo:

Make a large compound condition using many logical and comparison operators like this:

```
cout << (x== y || x <= y && x > y || x !=y) << endl;  // prints 0 (false)
```

Do not copy ours, make your own!

Precedence

Precedence forces which operation is to be one first. Multiplication and division has precedence over addition and subtraction.

```
int x = 3 + 4 * 6 =  3 + 24 = 27
```

You can force which operation you want to do first using round brackets ( )

```
Int x = (3 + 4) * 6 = 7 + 6 = 13
```

Now the 3 + 4 get calculated first and the 6 is then multiplied.

In compound conditions || (or) has precedence over && (and)

You can force which operation gets performed first by using round brackets

```
cout << (x== y ||  (x <= y && x > y) || x !=y) << endl;  // true
```

Now the && operation will get evaluated first.

Bitwise Operators

Bitwise operators act on operands as if they were binary digits. It operates bit by bit. Binary numbers are base 2 and contain only 0 and 1's. Every decimal number has a binary equivalent.

Every binary number has a decimal equivalent. For example:

decimal 4 is binary 0100
decimal 10 is binary 1010.

Binary numbers are calculated from power of 2 weights right to left.

Exponent	7	6	5	4	3	2	1	0
Power of 2	128	64	32	16	8	4	2	1
Binary	0	0	0	0	1	0	1	0

To calculate the decimal equivalent of a binary number you just add up the weights where a 1 is present.

8 + 2 = 10

The bit wise operators actually do logic operations on two numbers just like the arithmetic operators add two numbers

These logical operations are represents by truth tables
The & (and) operator is only true if a and b inputs both are true
The | (or) operator is only true if either are true
The ^ (xor) operator is only true if both are a and b are different false if both a and b are same.

& (and)				**| (or)**				**^ (xor)**		
a	B	result		a	b	result		a	b	result
0	0	0		0	0	0		0	0	0
0	1	0		0	1	1		0	1	1
1	0	0		1	0	1		1	0	1
1	1	1		1	1	1		1	1	0

The & | and ^ bitwise operators actual return a calculate bit result accordingly to the above truth table just like the adding and subtraction operators.

You can type in the operation right inside the cout statement just like this:

```
cout << (10 | 4) << endl;  // prints out 14
```

Alternatively, you can use variables instead like this:

```
x = 10
y = 4
cout << (x | y) << endl;  // prints out 14
```

Operator	Description	Example	Result
&	Bitwise AND	10 & 4	1010 10 decimal & 0100 4 decimal; ------- 0000 0 decimal
|	Bitwise OR	10 | 4	1010 10 decimal | 0100 4 decimal; ------- 1110 14 decimal
~	Bitwise NOT	~10	-11 (1111 0101 in binary)
^	Bitwise XOR	10 ^ 4	1010 10 decimal ^ 0100 4 decimal; ------- 1110 14 decimal
~	Bitwise NOT	~10	~1010 10 decimal ------ 0101 5 decimal

Shift operators

The shift operators multiply by powers of 2 or divide by powers of 2. Multiplying is accomplished by shifting the bits left. Dividing by 2 is accomplished by shifting the bits right.

```
int x = 10;  (0000 1010 in binary)
int y = x << 2;   y = 10 * 2 * 2 = 40 (0010 1000 in binary)
int z = y >> 2;  z = 40 /2 /2 = 10  (0000 1010 in binary)
```

You can shift by any power of 2

Multiplying by powers of 2:

```
x = 4;
cout << x << end;    // 4
x = x << 2;
cout << x << end;' // 16    because  4 * 2 * 2 = 16
```

Dividing by powers of 2:

```
cout << x << end; // 8
x = x >>  2;
cout << x << end; // 16 /2 / 2 = 2
```

<<	Bitwise left shift	10 << 2	10 * 2 * 2 = 40 (0010 1000 in binary)
>>	Signed Bitwise right shift	40 >> 2	40 /2 / 2 = 10 (0000 1010 in binary)

Increment/Decrement Operators ++   - -

Increment operators **++** increment a variable value by 1 and decrement operators **--** decrement a value by 1.

They come in two versions, **prefix** increment/decrement value **before** or **postfix** increment/decrement value **after**.
Prefix Increment before : y = ++x ;

x is incremented then value of y is assigned the value of x

```
                x       y
              ------  ------
int x = 5       5       ?
y = ++x         6       6
```

x = 5;
cout << x << endl; // 5
y =++x;
cout << y << endl; // 6
cout << x << endl; // 6

<u>postfix increment after   y = x++</u>

The value of y is assigned the value of x and then x is incremented

```
                x       y
              ------  ------
int x = 5       5       ?
y = x++         6       5
```

x = 5;
cout << x << endl; // 5
y =x++;
cout << y << endl; // 5
cout << x << endl; // 6

prefix Decrement before   y = --x

x is decremented then value of y is assigned the value of x

```
                          x        y
                        ------   ------
          int x = 5       5        ?
          y = ++x         4        4
```

```
x = 5;
cout << x << endl;     // 5
y =--x;
cout << y << endl;     // 4
cout << x << endl;     // 4
```

postfix decrement after   y = x--

The value of y is assigned the value of x and then x is decremented

```
                          x        y
                        ------   ------
          int x = 5       5        ?
          y = x--         4        5
```

```
x = 5;
cout << x << endl;  // 5
y =x--;
cout << y << endl;  // 5
cout << x << endl;  // 4
```

Increment decrement operators are usually used stand alone to increment or decrement a variable value by 1.

```
x++;
x--;
```

Assignment Operators

Assignment operators are used in C++ to assign values to variables.
x = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left. There are various compound operators in C++ like x += 5 that adds to the variable and later assigns the same. It is equivalent to x = x + 5.

```
int x = 5;
x += 5;
cout << (x) << endl;  // 10
cout << "x + = 5 = " << x < endl;
```

Operator	Compound	Equivalent
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x – 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
^=	x ^= 5	x = x ^ 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
<<=	x <<= 5	x = x << 5
>>=	x >>= 5	x = x >> 5

Lesson 4 Homework to do Part 1

1. Print out if a number is even, using just a print statement and a arithmetic operator

2. Print out of a number is odd, using just a print statement and a arithmetic operator

3. Swap 2 numbers using a temporary variable. Print numbers before and after swapping.

4. Multiply a number by 8 using the shift operator. Print numbers before and after shifting.

5. Divide a number by 8 using a shift operator. Print numbers before and after shifting.

6. In a print statement, add 2 numbers together and check if they are less than multiplying them together.

7. In a print statement, add 2 numbers together and check if they are less than multiplying them together **and** greater then multiplying them together.

In a print statement, add 2 numbers together and check if they are less than multiplying them together **or** greater then multiplying them together.

string Operators

String operators operate on string objects. To use strings you need to add

```
#include <string>
```

on the top of your cpp file to use strings.

There are many string operations, most of them are method calls. Here are just a few of them:

```
// declare and assign string
string s1 = "hello";
string s2 = "there";

// join two strings together
string s3 = s1 + s2;
cout << s3 << endl;   // hellothere
```

```
  // get a character from string
  char c = s3[0];
   cout << c << endl;   // h

   // get length of a string
   unsigned int length = s3.length();
    cout << length << endl; // 10

  //get a sub string start index and length
  string s4 = s3.substr(0,5);
   cout << s4 << endl; // hello

 //get a sub string  from start index to end of string
  string s5 = s3.substr(5); // prints out last letters from index 5
  cout << s5 << endl; // there

  // add a character to a string
  string s6 = s3.substr(0,5)+ 'X' + s3.substr(5);
  cout << s6 << endl; // helloXthere

  // test if 2 strings are equal
 cout << (s1 == s2) << endl; // false

 // test if 2 strings are  greater
 cout << (s1 > s2) << endl; // false

 // test if 2 strings are less
 cout << (s1 < s2) << endl; // true
// test if 2 strings are less greater or equal
// -1 = less  0 = equal  1 = greater
cout << (s1.compare(s2)) << endl; // -1
 cout << (s2.compare(s1)) << endl; // 1

// get index of a char
int index = s6.find('X');
cout << index << endl; // 5
```

```
 // get index of a string
 int index2 = s6.find("there");
 cout << index2 << endl; // 6

// replace  a string start position and length
s6.replace(0,5,"goodbye");
cout << s6 << endl; // goodbyeXthere

 // remove a string start position and length
 s6.erase(0,7);
 cout << s6 << endl; // Xthere

// find and replace   start, length and  newstring
 s6.replace(s6.find("there"),5,"files");
cout << s6 << endl; // X files
```

Lesson 4 Homework to do Part 2

8. Make a string of your favourite word and replace the first letter with another letter, hint use substring or replace.

   Example : change "hello" to "jello"

9. Make a string of your favourite word and replace the last letter with another letter, hint use substring or replace

   Example : change "jello" to "jelly"

10. Make a string of your favourite word and remove the middle letter, hint use substring or replace.

    Example : change "jelly" to "jely"

11. Using **substr** replace the last letter with the first letter in a word, do not use a temporary. Hint use length and many substr's.

    Example : change "jely" to "yelj"

12. Compare any of the two above strings are equal, greater or smaller to each other.

13. Make a string of your favorite words and find the index of some of your favorite letters. Once you find the index of your favourite letters erase the letters then print out the string, see what new word you get!

Put all your homework in a file called homework4.cpp

LESSON 5 PROGRAMMING STATEMENTS

Programming statements allow you to write complete C++ Program. We have already looked at simple input, print and assignment statements. We now present you with branch and loop programming statements. Continue with C++ file Lesson4.cpp and try out all these branch loop statements one by one. Once you see the program execution you will understand how theses branch and loop statements work. You may also want to add some extra statements of you own.

Branch Control Statements

Branch control statements allow certain program statements to execute and other not.

The **if** branch **control** statements contain a condition using conditional operators from the previous lesson to direct program flow.

```
If (condition)
    Statement(s)
```

When the condition is evaluated to be **true** the statements belonging to the if statement execute. An if statement is a one-way branch operation.

```
// if statement
x = 5;
if (x == 5)
{
     cout <<  "x is 5" << endl;
 }
```

```
x is 5
```

If you have more than one statement in your **if** statement then you need to use curly brackets so all statements get executed.

```
if (x >= 5)
        {
        cout <<  "x is 5" << endl;
        cout << "x is greater or equal 5" << endl;
        }
```

```
x is 5
 x is greater or equal to 5
```

Some people use curly brackets all the time.

We now add an else statement. An if-else control construct is a two-way branch operation.

```
If (condition)
     statements
else
      statements
```

```
// if – else statement
x = 2;
if (x == 5)
   cout << "x is 5" << endl;
else
   cout << "x is not 5" << endl;
```

```
x is not 5
```

We can also have additional else if statements to make a multi-branch.

```
// multi if else
x = 10;
if (x == 5)
   cout << "x is 5" << endl;
else if (x < 5)
```

```
   cout << "x less than 5" << endl;
else if (x > 5)
   cout << "x greater than 5" << endl;
```

```
x greater than  5
```

Our multi branch if-else can also end with an else statement.

```
// multi if-else else
x = 5;
if (x < 5)
        cout << "x less than 5" << endl;
else if (x > 5)
        cout << "x greater than 5" << endl;
else
        cout << "x is 5" << endl;
```

```
x is  5
```

If-else statements may be replaced by a switch statement. A switch statement is considered an organized if-else statement. It is a little limited since if can only handle equals. When the case values matched the switch value the statements in the case execute. The **break** keyword exits the switch statement. The default statements are executed of there is no match.

```
// switch statement
        x = 2;
        switch(x)
        {
                case 1:
                        cout <<  "x is 1" << endl;
                        break;

                case 2:
                        cout << "x is 2" << endl;
                        break;
                case 3:
                        cout << "x is 3" << endl;
                        break;
                default:
                        cout << "x is " <<  x << endl;
                        break;
        }
```

```
x is  2
```

if statements can also be nested to make complicated conditions simpler.
You need to add curly brackets to avoid confusion for which statements will be executed.

```
// nested if statement
x = 5;
if (x >= 0)
{
   if (x > 5)
      cout <<  "x greater than 5" << endl;
   else
      cout << "x less than equal 5" << endl;
}
cout << "I like C++ Programming" << endl;
```

```
x less than equal 5
I like C++ Programming
```

Loop Control Statements

Loop control statements allow program statements to repeat themselves.

while loop

The while loop allows you to repeat programming statements repeatedly until some condition is satisfied.

The while loop requires an initialized counter, a condition, program statements and then increment or decrement a counter. The while loop syntax is:

```
Initialize counter
while condition:
    statement(s)
    increment/decrement counter
```

When the condition is false the loop execution exits. While loops are used when you do not know how many items you have.
Here is a while loop that prints out the numbers 0 to 4;

```
// while loop
x = 0;
while (x <5)
{
   cout <<  x << endl;
   x++;
}
```

```
0
1
2
3
4
```

To do:

Make the while loop print out the numbers 1 to 5
Make a while loop that prints out the number 5 to 1 backwards.

do loop

The do loop also known as a **do-while** loop allows you to repeat programming statements repeatedly until some condition is satisfied. The condition is at the end of the loop, so the programing statements execute at least once.
The do loop requires an initialized counter, program statements, increment or decrement a counter and finally a condition.

The do loop syntax is:

```
Initialize counter
do{
    statement(s)
    increment/decrement counter
    }
    while condition;
```

When the condition is false the loop execution exits.
do loops are used when you do not know how many items you have.

Here is a do loop you can try out that prints out the number 5 to 1 backwards.

```
// do loop
x = 0;
do
{
   cout <<  x << endl;
   x++;
} while (x < 5);
```

```
0
1
2
3
4
```

To do:

Make the do loop print out the numbers 1 to 5
Make a do loop that prints out the number 5 to 1 backwards.

for Loop

Another loop is the **for** loop. It is much more automatic then the while loop but more difficult to use. All loops must have a counter mechanism. The for loop needs a start count value, condition, increment/decrement counter. When the condition is false the loop exits. For loops are used when you know how many items you have.

The for loop syntax is:

for (start_count_value,condition, increment/decrement_counter):
Statement(s)

Here we have a **for** loop to print out values 1 to 5, try it out.

```
// for loop
for (int i=;i<5;i++)
   cout <<  i << endl;
```

```
0
1
2
3
4
```

todo: Make the for loop print out the numbers 1 to 5

Here is a for loop that counts backwards using a negative increment

```
// for loop counting backward
for (int i=4;i>=0;i--)
   cout << i << endl;
```

```
4
3
2
1
0
```

todo: Make the **for** loop prints out the number 5 to 1 backwards.

Nested for loops

Nested for loops are used to print out 2 dimensional grids by row and column.

```
// nested for loop
for (int r=0;i<5;r++)
{
   cout << r << " : ";
   for (int c =0; c < 5; c++)
   {
      cout <<  c << " ";
  }
   cout << endl;
}
```

```
1 : 1 2 3 4 5
2 : 1 2 3 4 5
3 : 1 2 3 4 5
4 : 1 2 3 4 5
5 : 1 2 3 4 5
```

To do: change the rows and columns and see what you get

Loops can also be used to print out characters in a string variable

```
// print out characters in a string
s = "Hello";
for (unsigned int i=0;i<s.length();i++)
{
        cout <<  s[i] << endl;
}
cout << endl;
```

```
H
e
l
l
o
```

to do: printout string back wards

Replace all occurrences of a string with another string:

```
string s = "tomorrow";
string snew = "xxx";
while(s.find("o") != std::string::npos) {
    s.replace(s.find("o"),1,snew);

cout << s << endl;   //  txxxmxxxrrxxxw
```

Lesson 5 HOMEWORK TO DO:

Exam Grader

Ask someone to enter an exam mark between 0 and 100. If they enter 90 or above printout an “A”, 80 or above print out a “B”, 70 or above print out a “C”, 60 or above print out a “D” and “F” if below 60. Hint: use if else statements.

You can visualize a grade chart like this:

Mark Range	Exam Grade
90 to 100	A
80 to 89	B
70 to 79	C
60 to 69	D
0 to 59	F

Mini Calculator

Make a mini calculator that takes two numbers and a operation like - , +, * and /. Prompt to enter two number's and a operation like this:

```
Enter first number:  3
Enter second number: 4
Enter (+, -. *. /) operation:  +
```

Then print out the answer like this:

```
3 + 4 = 7
```

Hint: use a switch statement.
Use a while or do while loop so that they can repeatedly enter many calculations.
Terminate the program when they enter a letter like 'X' for the first number.

Triangle Generator:

Use nested for loops to print out a triangle using '*' like this:

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

Ask the user how many rows they want.

Hint: use 2 nested for loops, start with a square of stars

Enhanced Triangle Generator:

Use nested for loops to print out a triangle using '*' like this:

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

Ask the user how many rows they want.

Hint: use 2 nested for loops, start with a square of stars

Reverse a String

Reverse a String using a **while** loop or a **for** loop. Print the string before and after reversal.

Test if a number is prime

Make a function called **isPrime(x)** that tests if a number is print. In a loop divide the number between 2 to number-1 (or 2 to square root of number+1. For square root use:

```
x = (int)sqrt(n);
```

put

```
#include <cmath>
```

At the top of your program so that the C ++ compiler can recognize the **sqrt** function.

If the number can be divided by any of the divisors then the number is not prime, else it is prime. Print out the first 100 prime numbers.

The first 10 prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29

Print out all factors of a number

Make a function call factors(x) that will print out all the factors of a number. The factors of a number is all the divisors divided by the number evenly.
Example:

```
The Factors of 50 are:
1
2
5
10
25
50
```

Print out all prime factors of a number

Make a function call **prime_factors(x)** that will print out all the prime factors of a number. The prime factors of a number is all the prime number divisors divided by the number evenly.

Example: **12 = 2 × 2 × 3**

Following are the steps to find all prime factors.

```
0)  Enter a number n
1)  While n is divisible by 2, print 2 and integer divide n by 2
2)  In a for loop from i = 3 to square root of n + 1  increment by 2
          in a while loop while n is divisible by i
               print i
               integer divide n by integer i

3)  print n if it is greater than 2.
```

For square root use:

```
x = sqrt(n);
```

put

```
#include <cmath>
```

At the top of your program so that the C ++ compiler can recognize the **sqrt** function.

Make a Guessing game

Ask the user of your game to guess a number between 1 and 100. If they guess too high tell them "Too High". If they guess too low tell them they guess "Too Low". If they guess correct tell them "Congratulations you are Correct!". Play 10 games as a round. Keep track in an array how many tries each game took. At the end of 10 games in a table print out the tries for each game in the round. At the end of the table print out total score of all the game tries. For each round keep track of the lowest total score and inform the user if they beat the current lowest score or not. At the end of each round ask the user if they want to play another round of 10 games. You will need to first generate a random number to guess.

You can use this code to generate a random number:

```
// seed random number generator
srand((unsigned int)time(0));

// generate a random number
int number = rand() % MAX_NUMBER + 1;
```

Where **MAX_NUMBER** is a constant placed at the top of your program.

```
const int MAX_NUMBER = 100;
```

Also make another constant **MAX_GAMES** for the number of games to play.

```
const int MAX_GAMES  = 10;
```

You will need to include the following at the top of your program, for the compiler to recognize the **srand(), rand()** and **time()** functions.

```
#include <ctime>
#include <cstdlib>
```

You should have functions to print a welcome message explaining how to play the game, generate a random number, get a guess from the keyboard, check if a guess is correct and print out the game scores. The main function should just call your functions in a loop. Call your cpp file Homework5.cpp or GuessingGame.cpp

Object Oriented Guessing Game

Make a Guess Game class that will keep track of the guess number and tries per game. You should have a methods to generate a random number, check if a guess is correct, too low or too high and return the score per game. The main function should just handle inputs from the keyboard and printing output to the console. The GuessGame class should not handle any input and output, and is used, mainly to store data. The main function would instantiate a new GuessingGame object per round. The main would have an array of 10 Game Objects. After all games have been played print out the scores of each game and the average game. Also print out the game with the best score. Call your cpp file Homework5b.cpp or GuessingGame2.cp

LESSON 6 ARRAYS

Arrays store many sequential values together. We have one dimensional arrays and multi dimensional arrays. One dimensional arrays are considered a single row of values having many columns. You can visualize a one-dimensional array as follows. Array are sequential values under a common name.

value1	value2	value3	value4	value5

We declare and initialize 1 dimensional int array of size 5 as follows.
The size of the array is enclosed inside the square brackets.

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

1	2	3	4	5

We next declare a one-dimensional array of size 5 not initialized with values.

```
int a2[5];
```

In this situation you need to assign values separately as follows:
Arrays locations are assigned by an index. All indexes start at 0.

```
a2[0] = 1;
a2[1] = 2;
a2[2] = 3;
a2[3] = 4;
a2[4] = 5;
```

1	2	3	4	5

Arrays locations are also retrieved by an index. Index starts at 0

```
        int x = a[0];
        cout  << x << endl;  // 1
```

We can also allocate memory for arrays when the program is running;

```
int* a3 = new int[5];
```

a3 is known as a pointer because it holds the address of the allocated memory for the array. (points to the allocated memory)

You also access the allocated memory by index

```
a3[1] = 2;
int x = a3[1];
cout  << (x) << endl;  // 2
```

or by the pointer dereference operator *

```
*(a3+1) = 2;
int x = *(a3+1)
cout  << (x) << endl;  // 2
```

Once you are finished using the allocated array you need to reclaim the memory, so other programs can use the memory

```
delete[] a3;
```

We can use for loops to print out values in array.

// print out values in a 1 dimensional array

```
for (int i=0;i<5;i++)
{
   cout  << a2[i] << " ";
}
cout <<  endl;
```

```
1 2 3 4 5
```

Two-dimensional arrays

Two-dimensional arrays have grid of rows and columns. A two-dimensional array having 3 rows by 4 columns is visualized as follows:

Row 1	column 1	column 2	column 3	column 4
Row 2	column 1	column 2	column 3	column 4
Row 3	column 1	column 2	column 3	column 4

Here we declare and initialize a two-dimensional int array. We specify the number of rows and columns inside square brackets..

```
int b[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

1	2	3	4
5	6	7	8
9	10	11	12

We can also allocate memory for a 2 dimensional array. We first make a 1 dimensional array of int pointers for the number of rows.

```
        int** b2 = new int*[3];  // declare number of row pointers
```

We have declared a 1-dimensional array to hold int pointers. Each row pointer will point to a additional 1-dimensional array of int columns

```
b2[0] = new int[4];
b2[1] = new int[4];
b2[2] = new int[4];
```

We next assign values to the two-dimensional array by row index and column index . The row index first and the column index second. The row and column index's start at 0;

```
                        b2[2][3] = 11;
```

We retrieve values from the two-dimensional array also by row index and column index. The row index is first and the column index second.

The row and column index's start at 0;

```
        x = b2[2][3];
        cout << x << endl;  // 11
```

The row index and column index of a two-dimensional array can be visualized as follows. The row index is first and the column index second. The row and column index's start at 0;

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

```
// print out values in a two-dimensional array
        for (int r=0;r < 3; r++)
        {
           for (int c=0;c < 4; c++)
           {
                cout << b[r][c] << " ";
           }
                 cout << endl;
        }
```

```
1 2 3 4
5 6 7 8
9 10 11 12
```

Delete memory from an allocated 2-dimensional array.

We first delete memory for each row and then delete memory for the array storing the row pointers.

```
for (int r=0;r < 3; r++)
        {
        delete[] b2[r]
        }

delete[] b2;
```

LESSON 6 HOMEWORK

Question 1

Make an array of 10 numbers 1 to 10, print out the numbers in the array, then add up all the numbers and print out the sum.

Question 2

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Ask the user of your program to enter a number in the array. Search for the number in the array and report if it is found or not found.

Question 3

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Reverse all the numbers in the array. Print out the reversed array.

Question 4

Make a 2 dimensional array of 3 rows and 3 columns. Fill the 2 dimensional array with numbers 1 to 9. Add up the sum of all rows, and print the sum at the end of each row. Add up the sums of all columns, and print the sums at the end of each column.

Your output should look like this.

```
 1  2  3 : 6
 3  4  5 : 11
 6  7  9 : 22
--- --- ---
10 13 17
```

Question 5

Make an array to hold 10 numbers.
Generate 1000 random numbers between 1 and 10.
Keep track of the random numbers counts generated in your array.
Print out the counts of the numbers 1 to 10;
Print out the numbers with the smallest and largest count and number.
Print out the number of even and odd number counts and number.
You can make a random number like this:

```
#include <ctime>
#include <cstdlib>

srand((unsigned int)time(0));

int x = (rand() %10) + 1;
```

Put all answers in a cpp file called homework6.cpp

LESSON 7
COPY CONSTRUCTORS, ASSIGNMENT OPERATORS AND OVERLOADING

We continue with our Person class from previous lessons having a name and age.

```
class Person
    {
    private:
       string name;
       int age;

    public:

       Person(); // default constructor
       Person(string name, int  age); // parameter constructor
       string getName();// return  name
       void setName(string name); // assign name
       int getAge();// return age
       void setAge(int age);    // assign age
       string toString();    // return person info as a string
    };
```

const

The const keyword means cannot change a value once it is set.

```
const int SIZE = 100;
```

The constant variable **SIZE** is set to the value 100, and cannot be changed.

Object user data types can also be declared constant.

```
const Person p;
```

This means the variables of the person values cannot be changed.

Functions can also be **const**, especially getters.

```
string getName() const;
```

This means you cannot change the variables defined in the Person class that are inside the **getName()** method**.**

To do:

Make all your getters in your Person and Student class to be const.

Constructor with Default Parameters

A constructor with default constructor acts as both as an default constructor and a initializing constructor. Each parameter in a constructor with default values gets a default value.

```
Person (strong name="", int age=0);
```

You do not put the default values in the implementation, just in the header.

```
Person::Person(string name, int age)
{
	this->name = name;
	this->age = age;
}
```

Important: When you have a Constructor with default values you ,must remove the default constructor.

Copy Constructors

A copy constructor allows you to copy the values of an existing object. You make a copy constructor by supplying a constructor with a reference to an existing object.

We declare a copy constructor for a Person class defined a previous lesson like this:

```
Person(const Person& p);
```

We define a copy constructor for our Person class like this:

```
Person::Person(const Person& p)
{
    name = p.name;
    age = p.age;
}
```

The **const** key word means the parameter p Person object passed to the copy constructor cannot be modified. This means you are not allowed to change the values of the Person object p. We pass Person object p by reference using the & (ampersand). Pass by reference means the copy constructor receives a fixed address to the passed Person object. A reference is different from a pointer. A pointer stores an address that can be changed. The address that the pointer is pointing to, can be changed by the program, where as the address of a reference cannot be changed and is said to be a fixed address.

You would use a copy constructor like this:

First make some Person object.

```
Person p("Tom",24);
```

Make a second Person object p2 using the copy constructor.

```
Person p2(p);
```

Person object p2 will now have the values of Person object p;

If you print out the object using the toString() method from p2 the output would look like this:

```
Nice to meet you Tom
Tom You are 24 years old
```

Our Person class definition with a copy constructor would now look like this.

```
class Person
    {
    private:
       string name;
       int age;

    public:

       Person(string name="", int  age=0); // parameter constructor
       Person(const Person & p); // copy constructor
       string getName() const;// return  name
       void setName(string name); // assign name
       int getAge()const;// return age
       void setAge(int age);    // assign age
       string toString();    // return person info as a string
    };
```

To Do:

Update your Person.h and Person.cpp files to include a copy constructor. Make a new cpp file call Lesson5.cpp and make a Person object p with some hard coded values. Then print out the person object details. Next copy the Person objects p using the copy constructor. Now print out the copied object p2. The outputs should be the same. Lastly using the set name and age functions on copied Person object p2 change the name and age. Now print out the p and p2 again. The output should be different now because they are both separate Person objects.

Assignment operator =

An assignment operator lets you assign or copy another object to an existing object. The difference between a copy constructor and an assignment operator is that an assignment operator copies an **existing** object to an **existing** object where as copy constructor copies an **existing** object to a **new** object. For convenience the assignment operator uses the operator assignment symbol '='.

This is the same assignment operator used to assign a value to a variable like:

```
int x;
x  = 5;
```

The only difference is we will be assigning an object values to another object.

```
Person p;
p = Person("Tom",24);
```

We declare an assignment operator for a Person class like this:

```
Person& operator = (const Person& p);
```

We define an assignment operator for the Person class like this:

```
Person& Person::operator = (const Person& p)
{
    if ( this != &p )
        {
            name = p.name;
            age = p.age;
        }
return *this;
}
```

The assignment operator is similar to the copy constructor, in that they both receive a **const Person** object passed by reference, and they both assign values to the receiving object which is **this**. The keyword **this** is a pointer to the current object variables being acted upon.

We do not want the current object to copy itself so the following if statement is used.

```
If(this != &p)
```

The if statement means do not execute the following statements if the current object address is equal to the address of the receiving Person object p. The variable **this** is a pointer to the current object and &p is the address of the receiving object. The & means address of a variable or address of object variable like p in our case.

If they are different Person object addresses then the if statement is executed and the current object will be assigned new values.

```
If(this != &p)
    {
        name = p.name;
        age = p.age;
    }
```

Note a reference to the current executing object is returned. A reference is return for convenience so that other objects may want to be assigned to in a chain like this.

```
    p3=p2=p1;
```

You use an assignment operator like this

First make some Person object.

```
    Person p("Tom",24);
```

You first need to declare a Person object before you can use the assignment operator. If you do not then the copy constructor is called instead.

```
    Person p2;
```

Next assign Person p to Person p2 using the assignment operator.

```
    p2 = p;
```

Person object p2 will now have the values of Person object p;

If you print out the object using the toString() method from p2 the output would look like this:

```
Nice to meet you Tom
Tom You are 24 years old
```

Our Person class definition with an assignment operator would now look like this.

```
class Person
     {
     private:
        string name;
        int age;

     public:

        Person(string name="", int  age=0); // parameter constructor
        Person(const Person & p); // copy constructor
        Person& operator= (const Person & p); // assignment operator
        string getName()const;// return  name
        void setName(string name); // assign name
        int getAge()const;// return age
        void setAge(int age);    // assign age
        string toString();    // return person info as a string
        };
```

To understand the assignment operator think that the = sign is just a method name

p2 = p;

which really means

p2.=(p); or **p2.operator=(p);**

Which assign the values of p to p2;

Operators are used for convenience because we want to write a program that is more natural to use like this:

```
p2 = p;
```

Which assigns the values of person object p to person object p2;

To Do:

Update your Person.h and Person.cpp files to include an assignment operator. In your cpp file called Lesson5.cpp make a Person object p with some hard coded values. Then print out the person object details. Next assign the Person object p to p2 using the assignment operator. Now print out the copied object p2. The Output should be the same. Lastly using the set name and age functions on copied Person object p2 change the name and age. Now print out the p and p2 again. The output should be different now because they are both separate Person objects.

Equal operator ==

The equal operator == return true if 2 objects have equal values otherwise false. This is the same as using the variable equal = operator used to check if two variables are equal like:

```
int x = 5;
int y = 4;
if( x == y)
```

The only difference is we will be checking objects if they contain equal values variables.

```
Person p1 = Person("Tom",24);
Person p2 = Person("Sue",32);
if(p1 == p2)
```

We declare an equal operator for a Person class from like this:

```
bool operator ==(const Person& p);
```

We define an equal operator for the Person class like this:

```
bool Person::operator== (const Person& p)
{
     if ( name == p.name &&  age == p.age)
           return true;
    else
           return false;
}
```

The equal operator is similar to the assignment operator, in that they both receive a **const Person** object passed by reference. The difference is the equal operator return a bool data type bring true or false.

You use an equal operator like this:

First make some Person object.

```
Person p("Tom",24);
```

Then make a second Person object p2.

```
Person p2("Sue",32);

if( p == p2 )
        cout << "p is equal to p2" << endl;
else
        cout << "p is not equal to p2" << endl;
```

```
p is not equal to p2
```

Our Person class definition with an equal operator would now look like this.

```
class Person
    {
    private:
       string name;
       int age;

    public:

       Person(string name="", int  age=0); // parameter constructor
       Person(const Person & p); // copy constructor
       Person& operator= (const Person & p); // assignment operator
       bool operator== (const Person & p); // equal operator
       string getName()const;// return  name
       void setName(string name); // assign name
       int getAge()const;// return age
       void setAge(int age);    // assign age
       string toString();    // return person info as a string
    };
```

To understand the equal operator think that the == sign is just a method name.

```
If (p2 == p)
```

which really means

if(p2.==(p)) or **if(p2.operator==(p))**

Check's if the values of p are equal to the values of p2;
We use operators for convenience because we want to write a program that is more natural to use like:

```
If (p2 == p)
```

Check's if the values of p are equal to the values of p2.

To Do:

Update your Person.h and Person.cpp files to include an equal operator. In your cpp file called Lesson5.cpp make a Person object's p and p2 with some hard coded values. Then print out the each person object's details. Use the equal operator to test if both objects contain the same value. You may also want to use a copy constructor and assignment operator for some additional testing.

C++ Overloaded Operators

The following chart is a list of available C++ operators. They are called overloaded operators because they overloads the arithmetic operators +-*/% <> etc.

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

You may want to try a few out for curiosity.

Friend functions

Friend functions allow other functions not belonging to a class to access the private variables of some class.

A friend function is declared with the **friend** keyword in a class to let the compiler know this function can access the private variables of this class. The friend function code is defined outside a class. Friend functions are just ordinary stand alone functions, not belonging to any class.

The << and >> operator's used in cout and cin can also be overloaded so that the Person class can use them to print out or receive info from your class.

A friend function takes two parameters because the friend function does not belong to a class and therefore does not have a pointer pointing to the current object.

A friend function to overload the << operator to print out the contents of a Person object would be declared like this.

```
friend ostream& operator << (ostream& out, const Person& p);
```

The friend function code is usually put in the cpp file where the class code is implemented. Sometimes you may need to put the friend function code in the h file to avoid compilation errors. Friend function may be difficult to implement on some C++ compilers.

The code definition for a friend operator << output function for the person class is as follows:

```
ostream& operator << (ostream& out, const Person& p)
   {
   out << "Nice to meet you " << p.name << endl;
   out << p.name << " You are " <<  p.age <<  " years old";
   return out;
   }
```

The operator << gets an ostream object for outputting. It may output the contents to a console screen or a file depending on what is passed to it. The person object is passed by const reference so that the original person objects is received and not a copy. Pass by reference is more efficient and less overhead then pas by value. The **const** keyword again means the contents of the person object cannot be changed.

You use a friend function like this:

```
        Person p("tom",24);
        cout << p << endl;
```

The ostream object receives a cout object so that the person info is printed on the console scream A reference to the ostream object is returned. A reference is return for convenience so that other objects may want to be outputted to in a chain like this.

```
cout  << p1 << p2 << p3 << endl;
```

A friend function to overload the >> input operator to get info contents for a Person object would be declared like this.

```
friend istream& operator >> (istream& in, Person& p);
```

The code definition for a friend operator >> input function for the person class is as follows

```
istream& operator >> (istream& in, Person& p)
  {
     cout << "What is your name? ";
     in >> p.name;
     cout << "How old are you? ";
     in >> p.age;
     return in;
  }
```

The operator >> input function gets an istream object for inputting. It may input the contents from a keyboard or from a file depending on what is passed to it. The person object is passed by reference so that the original person object can be updated with new values. You use a friend function like this:

```
Person p;
cin >> p;
```

The stream object receives a cin object so that the person info is read from the keyboard. A reference to the istream object is returned. A reference is return for convenience so that other objects may want to be input in a chain like this.

```
cin   >> p1 >> p2 >> p3;
```

Our Person class definition with a friend function operator << and operator >> would now look like this.

```
class Person
    {
    private:
       string name;
       int age;

    public:

       Person(string name="", int  age=0); // parameter constructor
       Person(const Person & p); // copy constructor
       Person& operator= (const Person & p); // assignment operator
       bool operator== (const Person & p); // equal operator
       friend ostream& operator << (ostream& out, const Person& p);
       friend istream& operator >> (istream& in, Person& p);
       string getName()const;// return  name
       void setName(string name); // assign name
       int getAge()const;// return age
       void setAge(int age);    // assign age
       string toString();    // return person info as a string
    };
```

The code for the friend functions would be in the Person cpp file
Or can be in the Person.h depending what your C++ compiler prefers.

To Do:

Update your Person.h and Person.cpp files to include the output and input friend functions . In your cpp file called Lesson6.cpp make a Person object p, with some hard coded values. Then print out the person object details using the output operator <<. Then use the input operator >> to get new values from the keyboard. Lastly print out the person object details using the output operator <<.

LESSON 7 HOMEWORK Question 1

Update your Profession class from previous homework that stored a profession title and salary. Add a copy constructor, assignment operator and equals operator, that would compare the professions title and salary. Also add output << and input >> friend stream operators to your Profession class. Update your main function to use the copy constructor, assignment operator and equals operator.

You no longer need to use the enterProfession and enterSalary functions.
Use the output << and input >> stream operators instead..

Add **const** to all your getters.

You can put everything in a file called Homework7.cpp.

Calling constructors from other constructors

To avoid code repetition one constructor may want to call another constructor. A constructor always needs to validate inputs, if one input is invalid it may want to initialize all variables you a default value. In this case a parameter constructor may want to call the default constructor to avoid repetition code.

For example if the passed name parameter is empty then you would call the default constructor to set all variables to a default value like this:

```
Person::Person(string name, int  age)
{
   if(name=="")
   {
      *this = Person();
   }
   else
   {
   this->name = name;
   this->age = age;
   }
}
```

***this** is used assign the values from the default constructor by way of the assignment operator. Since this points to the current object, the current object values gets initialized to default values.

It is difficult to call a constructor from the assignment operator but to save code you may call the assignment operator from the copy constructor instead like this

```
Person::Person(const Person& p)
{
   operator =(p);
}
```

We just call the assignment operator just like a normal function call using the operator keyword and the assignment operator function name =.

Other alternatives for saving code are to use default parameters or a set function.

Default parameters let you specify default values for you parameter values when you define the methods in your class. They work right to left. So it means the last parameters need to be initialized first. To specify default parameter for the person parameter constructor you would do this.

```
Person(string name="", int  age=0);
```

Now you do not need the default constructor code any more.

You can now make person objects like this:

```
Person p;
```

Or with argument values like this:

```
Person p("tom",24);
```

Just by using one constructor the parameter constructor and default parameter values.

Note: Default parameter initialization values only go on the method definition not on the method implementation.
Set functions are very handy to use and avoid duplication of code in the constructors and assignment operator.

A set function is similar to a constructor but is just a normal method that can be called any time to assign values to an object. The set function is very similar the setters except they have more than 1 parameter value.

```
Person::set(string name, int  age)
{
   if(name=="")
   {
      this->name = "";
      this->age = 0
   }
   else
   {
   this->name = name;
   this->age = age;
   }
}
```

Note this set function validates and calls the default constructor to initialize the objects values to a default values. You can use the set function in the parameter constructor like this:

```
Person::Person(string name, int  age)
{
  set(name, age);
}
```

You can use the set function in the assignment operator like this:

```
Person& Person::operator = (const Person& p)
{
     if ( this != &p )
     {
    set(p.name, p.age);
     }
return *this;
}
```

Many people like and use set functions because you can have common code for validation and assigning an object values**.**

Things to do:

Update your Person.h and Person.cpp files as follows. First call the default constructor from the parameter constructor. In your cpp file called Lesson5.cpp make a Person object p, with an empty name and any non-zero age. Print out p to verify if the age was set to 0. Next call the assignment operator on the copy constructor. Make a non empty Person object p2 and assign p2 to p; Print out p and p2 to verify if they are the same values. Now include a default parameter constructor. You can remove the default constructor since it is no longer needed. Make an empty Person object and non empty person objects using the default parameter constructor. Print out them both and verify one is empty and one is not empty. Finally make the set method and include one in the default parameter constructor and in the assignment operator. Make some empty and non empty Persons and print them out to verify valid operation. Then use the set function on the empty person object with some non empty values and print out the person object to verify valid operation.

Adding the copy constructor, assignment operator, equals operator and stream operators to a derived class

We will use the student class from previous lessons.

Here is the student class definition with the additional contructors and operators

```
class Student: public Person {

 private:
    string idnum; // student id number

public:
   Student(); // default student
   Student(string name, int  age, string idnum); // initialize Student
   string getID()const;       // return idnum
   void setID(string idnum); //assign idnum
   string toString();    // return student info string
};
```

Derived Student class Constructor with Default Parameters

A constructor with default constructor acts as both as an default constructor and a initializing constructor. Each parameter in a constructor with default values gets a default value.

```
Student (string name="", int age=0, string idnum=””);
```

You do not put the default values in the implementation, just in the header.

```
Student::Strudent(string name, int age, string idnum)
: Person(name, age)
{
       this->idnum = idnum;

}
```

Important: When you have a Constructor with default values you ,must remove the default constructor.

Derived Student class Copy constructor

The derived copy constructor is similar to a standard copy constructor. We use the base class **Person** and derived class **Student** from previous Lessons.

Student(const Student& s);

The only additional thing we need is to initialize the base class copy constructor with an initializer and initialize the **idnum** as follows. The compiler will extract the Person from the Student and call the Person constructor to initialize the Person.

```
Student::Student(const Student& s)
    : Person(s)
    {
       this->idnum = s.idnum;
    }
```

Since the Student class is derived from the Person class the Person part of the Student class is passed to the Person base class copy constructor automatically.
You would use the student copy constructor like this:
First make some Student object.

```
Student s("Tom",24, "S1234");
```

Then make a second Student object s2 using the copy constructor.

```
Student s2(s);
```

Student object s2 will now have the values of Student object s;

If you print out the object using the toString() method from s2 the output would look like this:

```
Nice to meet you Tom
Tom You are 24 years old
Your student id number is S1234
```

To Do:

Update your Student.h and Student.cpp files to include a copy constructor. Make a new cpp file call Lesson5.cpp and make a Student object s with some hard coded values. Then print out the student object details. Next copy the Student objects s using the copy constructor. Now print out the copied object s2. The outputs should be the same. Lastly using the set idnum on copied Student object s2 change the idnum. Now print out the s and s2 again. The output should be different now because they are both separate Student objects.

Derived Student class Assignment operator =

The derived class assignment operator is similar to the standard assignment operator, the only additional thing we need to do is call the assignment operator of the base class.

We declare an assignment operator for a Student class like this:

```
    Student& operator=(const Student& s);
```

We define an assignment operator for the Student class like this:

```
Student& Student::operator=(const Student& s)
  {
    if(this != &s)
    {
      // call the assignment operator from the Person class
      Person::operator=(s);
      this->idnum = s.idnum;
    }
    return *this;
  }
```

Note: Inside the Student assignment operator we have called the Person base class assignment operator like this:

```
        Person::operator=(s);
```

You use an assignment operator like this:

First make some Student object.

```
Student s("Tom",24, "S1234");
```

You first need to declare a Student object before you can use the assignment operator. If you do not then the copy constructor is called instead.

```
Student s2;
```

Then assign Student s to Student s2 using the assignment operator.

```
s2 = s;
```

Student object s2 will now have the values of Student object s;

If you print out the object using the toString() method from s2 the output would look like this:

```
Nice to meet you Tom
Tom You are 24 years old
Your student id number is S1234
```

To Do:

Update your Student.h and Student.cpp files to include an assignment operator. In your cpp file called Lesson7.cpp make a Student object s with some hard coded values. Then print out the Student object details. Next assign the Student object s to s2 using the assignment operator. Now print out the copied object s2. The outputs should be the same. Lastly using the set id function on the copied Student object s2 change the id num. Now print out the s and s2 again. The output should be different now because they are both separate Student objects.

Derived Student class Equal operator ==

The equal operator == returns true if 2 Student objects have equal values otherwise false.

We declare an equal operator for a Student class from like this:

```
bool operator ==(const Student& s);
```

We define an equal operator for the Student class like this:

```
bool Student::operator ==(const Student& s)
{
// check if Person name and age equal and student id is equal
    if ( Person::operator==(s) && idnum == s.idnum)
        return true;
    else
        return false;
}
```

Note: Inside the Student operator= method we have called the base equal operator like this:

```
Person::operator==(s)
```

You use an equal operator like this

First make some Student object.

```
Student s("Tom",24, "s1234");
```

Then make a second Student object s2.

```
Student s2("Sue",32, "s5678"););
```

You can use like this:

```
If( s == s2 )
    cout << "s is equal to s2" << endl;
else
    cout << "s is not equal to s2" << endl;
```

```
s is not equal to s2
```

To Do:

Update your Student.h and Student.cpp files to include an equal operator. In your cpp file called Lesson5.cpp make a Student object's s and s2 with some hard coded values. Then print out the each Students object's details. Use the equal operator to test if both objects contain the same value.

Friend derived class functions

Friend functions can also be used with derived classes.
A friend function to overload the << operator to print out the contents of a Student object would be declared like this.

```
friend ostream& operator << (ostream& out, const Student& s);
```

The code definition for a friend operator << output function for the Student class is as follows

```
ostream& operator << (ostream& out, const Student& s)
   {
   out << (Person&)s << endl; // Call Person << stream operator
   out << "Your student id is " << s.idnum << endl;
   return out;
   }
```

Note: we call the base class output stream operator like this:

```
  out << (Person&)s << endl;
```

We need to typecast the s to a Person reference so the it prints out the Person info only.

You use an output stream friend function like this:

```
        Student s("Tom",24,"S1234");
        cout << s << endl;
```

A friend function to overload the >> input operator to get info contents for a Student object would be declared like this.

```
        friend istream& operator >> (istream& in, Student& s);
```

The code definition for a friend operator >> input function for the Student class is as follows

```
istream& operator >> (istream& in, Student& s)
   {
      in >> (Person&) s;
      cout << "What is your student id? ";
      in >> s.idnum;
      return in;
   }
```

Note: we call the base class input stream operator like this:

```
  in >> (Person&) s;
```

We need to typecast the s to a Person reference so the it access the Person info only.

You use the input stream friend function like this:

```
        Student s;
        cin >> s;
```

Our Student class definition now looks like this:

```
class Student: public Person {

 private:
    string idnum; // student id number

public:
   Student(string name=””, int  age=0, string idnum=””); // initialize Student
   Student(const Student& s); // copy constructor
   Student& operator=(const Student& s); // assignment operator
   bool operator ==(const Student& s) // equals operator
   friend ostream& operator << (ostream& out, const Student& p);
   friend istream& operator >> (istream& in, Student& p);
   string getID()const;       // return idnum
   void setID(string idnum); //assign idnum
   string toString();    // return student info string
};
```

To Do:

Update your Student.h and Student.cpp files to include the output and input friend functions . In your cpp file called Lesson5.cpp make a Student object s, with some hard coded values. Then print out the Student object details using the output operator <<. Then use the input operator >> to get new values from the keyboard. Lastly print out the Student object details using the output operator <<.

Additional things to do.

Change the Student parameter constructor to a default parameter constructor. make a set function for the Student class and the set function to the Student copy and assignment operator. You can call the base class set method like this.

```
        Person::set(s.getName(), s.getAge());
```

Make some empty and non empty Students and print them out to verify valid operation. Then use the set function on the empty student object with some non empty values and print out the student object to verify valid operation.

Lesson 7 Homework Question 2

From previous homework's update the Profession and Department classes. Add copy constructors, assignment = and equals == operators . Also add the << output and >> input stream friend operators. Replace the initializing constructor to use default parameters, then remove the default constructors . In the main method test all the new functions. Put the main method in a file called homework7.cpp.

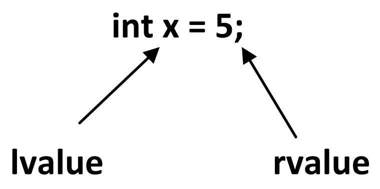
Lesson 8 Move Constructors and Move Assignment Operators

Move constrictors and move assignment operators move data from one object to another using a pointer to the same memory address. Moving data using the same memory address is more efficient then copy values from one memory address to the other. You can put all code in a file called Lesson8.cpp.

Rvalue and lvalue's

An **rvalue** is an expression that does not have any memory address, and an **lvalue** is an expression with a memory address.

An **lvalue** is a variable in the left side of an assignment statement that stores a values where a **rvalue** is a value on the right side of an assignment statement that does not have a memory address like a constant.



You cannot assign a variable to a constant:

```
5 = x
```

You cannot assign a constant to a reference because a constant does not have an address,

```
// int& x = 5;
```

But you can move a lvalue to a rvalue reference using the move operator &&

```
int&& x = 5

cout << x << endl;  // 5
```

Move Constructor

A **move constructor** allows the resources owned by an **rvalue** object to be moved into an **lvalue** without creating a copy. Usually the object has allocated memory to be moved by pointer from one pointer to another.

In this situation the pointer in the **lvalue** object is pointed to by the **rvalue** memory object. The pointer in the **rvalue** object then is set to nullptr, indicating that the memory has been moved.

Our **rvalue** and **lvalue** will be an MyArray class having a pointer memory and a size variable. Our move constructor is a follows:

```
// move constructor
 MyArray ( MyArray && arr){

   // assign memory address from other array
   this->ptr = arr.ptr;
   this->size = arr.size;

   // set other array ptr to null indicating it has been moved
   arr.ptr = nullptr;
   arr.size = 0;
 }
```

To use the move constructor you must use **std::move** cast.   The **std::move()** is a cast that produces an **rvalue-reference** to an object, to enable moving from it. The **std::move**(x) is equivalent to   **static_cast<T&&>(t);**

To us we the move constructor you first make the **lvalue** object

```
        MyArray  a;
```

Then make the rvalue object by calling the move copy constructor using the move operator cast and the lvalue object a.

```
        MyArray b=std::move(a);
```

Here is a **MyArray** class Example using a move constructor.

```
#include <iostream>

using namespace std;

const int DEFAULT_SIZE = 10;

// array class
class MyArray{
private:
  int *ptr; // pointer to memory
  int size; // size of memory

public:

  // default constructor
  MyArray(){

    cout << "Calling Default constructor" << endl;
    ptr = new int [DEFAULT_SIZE];
    size = DEFAULT_SIZE;
  }

  // initializing  constructor
  MyArray(int size){

    cout << "Calling Initializing constructor" << endl;
    ptr = new int [size];
    this->size = size;
  }

  // copy constructor
  MyArray( const MyArray & arr){

   cout << "Calling Copy constructor" << endl;
```

```
     // create memory for array
     this->ptr = new int[arr.size];
     this->size = arr.size;

     // copy values from other array
     for(int i=0;i<arr.size;i++)
     {
        ptr[i] = arr.ptr[i];  // copy values from arr
     }
    }

    // move constructor
    MyArray ( MyArray && arr){

      cout << "Calling Move constructor" << endl;

      // assign memory address from other array
      this->ptr = arr.ptr;  //  use memory from arr
      this->size = arr.size; // assign size from arr

      // set other array ptr to null indicating it has been moved
      arr.ptr = nullptr;
      arr.size = 0;
    }

     // destroy memory
     // note: nullptr cannot be deleted
     // so program will not crash deleting a nullptr
    ~MyArray(){
     // Destructor
     cout << "Calling Destructor"<<endl;
     delete[] ptr;
     size=0;
     }

};
```

Here is the main function:

```
int main() {

// make array
MyArray a;

// using copy constructor
// (not using move constructor)
cout << "not using move constructor" << endl;
MyArray b(a);

// using move constructor
cout << "using move constructor" << endl;
MyArray c=std::move(a);

return 0;
}
```

```
5
5
Calling Default constructor
not using move constructor
Calling Copy constructor
using move constructor
Calling Move constructor
Calling Destructor
Calling Destructor
Calling Destructor
```

Todo: print out which constructor got deleted

Move Assignment operator

The move assignment operator just moves the recourses from the other lvalue object and then signals the resources that they have been moved. In this situation before resources are moved from the other **lvaue** object the memory recourses in this object are deleted first.

Here is the standard assignment operator:

```
 MyArray& operator=(MyArray& arr)
 {
    cout << "Calling assignment operator" << endl;

   // don't copy self
    if(this != &arr)
    {
       // delete old memory
       delete[] ptr;
       Size=0;

       // make new memory
       arr.ptr = nullptr;
       arr.size = 0;

     // copy values from other array
     for(int i=0;i<arr.size;i++)
           {
              ptr[i] = arr.ptr[i];  // copy values from arr
           }

    }
    return *this;
 }
```

Here is the move assignment operator:

```
 // move Assignment operator
 MyArray& operator=(MyArray&& arr)
 {
    cout << "Calling Move assignment operator" << endl;
    if(this != &arr)
    {
       // point to other array values
       ptr = arr.ptr;
       size = arr.size;
       // set other ptr to nullptr indicating values have been moved
       arr.ptr = nullptr;
       arr.size = 0;
    }
    return *this;
 }
```

Here is the move assignment test code:

```
cout << "not using assignment operator" << endl;
MyArray d;
d = c;

// using move
cout << "using move assignment operator" << endl;
MyArray e;
e=std::move(c);
```

We first declare MyArray objects d and e then assign the c array to them so we call the assignment operator rather than the constructors. We use the c array because the memory in the a array has been previously deleted.

Here is the test program output:

```
Calling assignment operator
using move assignment operator
Calling Default constructor
Calling Move assignment operator
```

```
}
```

Using the Derived class move copy constructor

When you call the base move copy constructor you have to use **std:move** or else the non move copy constructor is called.

```
DerivedClass(DerivedClass&& other) : BaseClass(std::move(other))
 {
 }
```

Using Derived class move assignment operator

When you call the base move assignment operator you have to use **std:move** or else the non move assignment operator is called.

```
DerivedClass(DerivedClass&& other)
 {
        if(this != &other)
         {
             BaseClass::operator=(std::move(other));

         }

        return *this;
 }
```

Here is a class the inherits the MyArray class.

```
Class MySumArray: public MyArray
{
        private:
        int sum;

       // default constructor
       MySumArray():MyArray()
       {
             sum = 0;
       }
```

```
        // intializer constructor
        MySumArray(int size):MyArray(size)
        {
                sum = 0;
        }

           // copy constructor
          SumMyArray( const MyArray & arr):MyArray(arr)
         {

                sum = arr.sum;
           }

           // move constructor
           SumMyArray ( MyArray && arr):MyArray(std::move(arr))
         {

                 sum = arr.sum;
           }

// assignment operator
MyArray& operator=(MyArray& arr)
{
     if(this != &arr)
   {
              MyArray::operator=(arr);
              sum = arr.sum();

   }

     return *this;
}
```

```
// move assignment operator
MyArray& operator=(MyArray&& arr)
{
        if(this != &arr)
         {
             MyArray::operator=(std::move(arr));
              sum = arr.sum();

         }

      return *this;
}
```

Todo: Test the SumMyArray class in a main. Add a method to sum the values in the array and return them.

LESSON 9 Homework Question 1

Add the move copy and move assignment operator to the Person and Student class of the previous lesson. Test the copy, move copy, assignment and move assignment operators. Make sure in your main you call the move copy constructor and the move assignment operator using std::move operator.

Conversion operator

A conversion operator returns a value from a object. This may be a value or a calculated value. A conversion operator is similar to what a ordinary function that would do. Example we can have a conversion operator **operator int()** to return the sum of the calculated values.

```
// conversion operator
// return sum of a array values
operator int()
{
   int sum  = 0;
```

```
    for(int i=0;i<size;i++)
    {
       sum += ptr[i];
    }
    return sum;
 }
```

Calling conversion operator is similar calling a function except the right value is the object variable.

```
int total = e; // calling conversion operator

cout << total << endl;
```

overloading operator()

The **operator()** can be used for many things. We will use it to assign and retrieve values in the array and to clear the array.

```
// assign or retrieve values
 int& operator()(int i)
 {
    return ptr[i];
 }

 // retrieve values
 int operator()(int i) const
 {
    return ptr[i];
 }
```

```
// clear array
 void operator()()
 {
    for(int i=0;i<size;i++)
    {
       ptr[i]=0;
    }
 }
```

We would call the overloading operators like this

```
// assign a value
e(0) = 5;

// read a value
cout << e(0) << endl;

// clear array values
e();
```

using [] to access values in the dynamic array

Using [] brackets rather than ()

```
// assign retrieve values
 int& operator()(int i)
 {
    return ptr[i];
 }

 // retrieve values
 int operator()(int i) const
 {
    return ptr[i];
 }
```

```
// assign a value
e[0] = 5;

// read a value
cout << e[0] << endl;
```

Explicit constructor

An Explicit keyword prevents implicit conversions. An implicit conversion happens when the compiler does any automatic conversion it can do. In our case we have a initializing constructor that can be called multiple ways.

```
MyArray(int sz){
  cout << "Calling Initializing constructor" << endl;
  size = sz;
  ptr = new int [sz];
}
```

We can call the initializing constructor like this:

```
MyArray a(5)
```

or like this:

```
MyArray a = 5;
```

Using the explicit keyword would force that the constructor only use this way.

```
MyArray a(5)
```

Not this way:

```
MyArray a = 5;
```

To prevent this from happening we put the explicit keyword on the initializing size constructor.

```
    explicit MyArray(int sz){

      cout << "Calling Initializing constructor" << endl;
      size = sz;
      ptr = new int [sz];
     }
```

Now when we call:

```
        MyArray a = 5;
```

We get this compiler message:

```
error: conversion from 'int' to non-scalar type 'MyArray' requested
```

explicit means that we are not preventing the constructor to be called, it means to stop implicit conversion instead.

Note: we could still call the size constructor with a double since the double would be truncated to a int.

```
        MyArray a(5.5);
```

Stopping the default constructor and assignment operator from being called

You can mark the copy constructor and copy assignment operator as delete to stop it from being called.

```
MyClass (const MyClass&) = delete;
MyClass& operator= (const MyClass&) = delete;
```

In this situation you do not to write code for the copy constructor and copy assignment operator.

Lesson8 Homework Question 2

Add the conversion operator, overload() operator and add **delete** to the copy constructor and assignment operator of the MyArray class. Add the **explicit** keyword to the copy constructor and try to prevent some conversion. Add the test code to main.

LESSON 9   STL VECTORS, LISTS, SETS AND MAPS

STL stands for Standard Template classes. These template calls are used a lot in C++ programming. They provide expandable arrays known as vectors, link list known as list, expandable array the only store unique values called sets and a collection the stores name and value pairs known as map**.**

For the this lesson make a new C++ source file called Lesson6.cpp and in the main function type in the following programming statements for the following Vectors, Lists, Sets and Maps.

vectors

vectors are expandable arrays that store values. A vector needs to know what type of data type it will use. You need to specify the data type to be used inside diamond <> brackets like **<int>.** You will also need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using vectors in your program.

```
// To create an empty vector
vector<string> v;

  // Add a value to a vector
  v.push_back ("Cat");
  v.push_back ("Dog");
  v. push_back ("Lion");
  v. push_back ("Tiger");
```

We use the **size** method to print out the number of elements in a list.

```
// get and print the number of elements in an vector
unsigned int size = v.size();
cout <<  size  << endl;  // 5
```

You can get and print a value from a vector at a specified location by specifying an index.

```
        cout <<  v[0] << endl; // Cat
```

Change a value at a index:

```
        v[0] = "Zebra";

        cout <<  v[0] << endl; // Zebra
```

C++ does not print out the vector for you, so you need to do it your self using loops.

```
// Print out  array vector
for(unsigned int i=0;i<v.size();i++)
{
        cout << v[i] << " ";
}
```

```
Cat Dog Lion Tiger
```

A vector also has an **iterator** that can be used to print our vectors item values. An iterator will access the items stored in a vector sequentially one by one.

To use an iterator you will need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using **iterator** in your program.

The vector has function **begin()** that returns an iterator at the beginning of the vector and the function **end()** that returns an iterator at the end of the vector. The end of the vector contains no value but is just used to specify the end of the vector.

```
vector<string>::iterator itr;

 for(itr=v.begin(); itr!=v.end(); itr++)
            {
            cout << *(itr) << " ";
            }
 cout << endl;
```

```
Cat Dog Lion Tiger
```

search for an item in a vector

To search for a value we use the **find** function. You first specify a search range using the **begin()** and **end()** functions. They return iterators at the beginning of the list and the end of the list. The end of the list does not contain any value, but is used to specify the end of the vector. If the item is not found then the end iterator is returned. Find location of the animal.

```
itr = find(v.begin(), v.end(), "Lion");

 if (itr != v.end())
      cout << "found " << *itr << endl; //  found Lion
else
      cout << "did not find Lion "<< endl; // did not find Lion
```

Once you have an iterator pointing to your search value you can also change its value.

```
*itr2 = "Elephant";

cout << *itr << endl; // Elephant
```

If you do find the element then the itr points to the last value you specified, in our case the end of the vector. Iterator points to location in the vector so we can read it or change it.

We print out the list again:

```
// print list
for(itr2=v.begin(); itr2!=v.end(); itr2++)
        {
        cout << *(itr2) << " ";
        }
cout << endl;
```

```
Cat Dog Elephant Tiger
```

To use the **find** method you need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using **find** function in your program.

removing an item from a vector

You can remove an item pointed to by a iterator and using the erase function. This is a two step process;

Step 1: point to an item in the vector using the **find** method.

```
itr = find(v.begin(), v.end(), "Dog");
```

Step 2: remove the item using the **erase** function.

```
v.erase (itr);
```

```
// print out vector
 for(itr=v.begin();itr!=v.end();itr++)
          {
          cout << *(itr) << " ";
          }
 cout << endl;
```

```
Cat Elephant Tiger
```

You can also remove a value from a vector by specifying an iterator and a index

```
// remove from index 2 (Elephant)
v.erase (v.begin()+2);

// print out vector
 for(itr=v.begin();itr!=v.end();itr++)
           {
           cout << *(itr) << " ";
           }
  cout << endl;
```

```
Cat Tiger
```

To sort a vector you use the sort function. You first specify a sort range using the **begin()** and **end()** functions.

You need to put at the

To use the **sort** method you need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using **sort** function in your program.

To sort a vector ascending we use the optional **less** function.

```
// sort a vector  ascending
  sort(v.begin(), v.end(),less<string>());

   for(itr=v.begin();itr!=v.end();itr++)
           {
           cout << *(itr) << " ";
           }
  cout << endl;
```

```
Cat Dog Tiger
```

To sort a vector descending we use the **greater** function.

```
// sort a vector descending
   sort(v.begin(), v.end(),greater<string>());


    for(itr=v.begin();itr!=v.end();itr++)
            {
            cout << *(itr) << " ";
            }
   cout << endl;
```

```
Tiger Dog Cat
```

lists

Lists use link nodes to make a chain of values. A node is a memory cell that contain a value to store and a pointer to the next node. In case of the last node the next node points to NULL. When you add a value to a list it makes a new node and adds it to the end or beginning of the list. A **list** needs to know what type of data type it will use so you need to specify the data type to be used inside the diamond brackets **<int>.** You will need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using lists in your program.

```
// To create an empty list
 list<string> ls;

// Add a value's to end of a List
ls.push_back("Cat");
ls.push_back("Dog");
```

```
// Add a value's to start of a List
ls.push_front("Lion");
ls.push_front("Tiger");
```

You cannot access individual elements in a list so you must use an Iterator to print our list item values. An iterator will access the items stored in a list sequentially one by one.

To use an iterator you will need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using iterator in your program.

You print a list using the iterator as follows:

```
list<string>::iterator itr2;

for(itr2=ls.begin();itr2!=ls.end();itr2++)
        {
        cout << *(itr2) << " ";
        }
cout << endl;
```

```
Tiger Lion Cat Dog
```

We use the **size** method to print out the number of elements in a list.

```
// get and print the number of elements in an list
unsigned int size = ls.size();
cout <<  size  << endl; // 5
```

To search for a value we use the **find** function. You first specify a search range using the **begin()** and **end()** functions. They return iterators at the beginning of the list and the end of the list. The end of the list does not contain any value, but is used to specify the end of the list. If the item is not found then the end iterator is returned.

```
itr2 = find(ls.begin(), ls.end(), "Dog");

 if (itr != ls.end())
     cout << "found " << *itr2 << endl; //  found Dog
else
     cout << "did not find Dog" << endl; // did not find Dog
```

Once you have an iterator pointing to your search value you can also change its value.

```
*itr2 = "Puppy";

cout << *itr2 << endl; // Puppy
```

If you do find the element then the itr points to the last value you specified in our case the end of the list.

```
// print list
   for(itr2=ls.begin();itr2!=ls.end();itr2++)
          {
          cout << *(itr2) << " ";
          }
cout << endl;
```

```
Tiger Lion Cat Puppy
```

To use the **find** method you need to place

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

At the top of your .cpp or .h file so that the compiler will know that you are using **find** function in your program

To remove a item from the list by value we use an iterator and the advance function. The advance function will advance the iterator by a specify number. We will remove "cat" at index 2

We first point to the beginning of the list

```
itr2 = ls.begin();
```

Then advance the iterator by 2

```
advance(itr2,2);
```

To use the **advance** method you need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using **advance** function in your program.

We then use the erase function to remove the element pointed to by the iterator.

```
ls.erase(itr2);

// print list
 for(itr2=ls.begin();itr!=ls.end();itr++)
           {
           cout << *(itr2) << " ";
           }
  cout << endl;
```

```
Tiger Lion Puppy
```

A list has a built in **sort** method to sort a list

```
// sort a list  ascending
 ls.sort();

   for(itr2=ls.begin();itr!=ls.end();itr2++)
          {
          cout << *(itr2) << " ";
          }
  cout << endl;
```

```
Lion Puppy Tiger
```

A list has a built in **reverse** method to reverse a list

```
// sort a list descending
   ls.reverse();

   for(itr2=ls.begin();itr!=ls.end();itr++)
          {
          cout << *(itr2) << " ";
          }
  cout << endl;
```

```
Tiger Puppy Lion
```

A list has methods to retrieve front and back values:

```
// get value at front of list
 cout << ls.front() << endl;

 // get value at back of list
 cout << ls.back() << endl;
```

A list has methods to remove front and back values:

```
// remove a value from start of a List
ls.pop_front(); // Tiger
```

```
  // remove a value from end of a List
 ls.pop_back(); // Lion

  // print list
   for(itr2=ls.begin();itr2!=ls.end();itr2++)
          {
          cout << *(itr2) << " ";
          }
   cout << endl;
```

```
Puppy
```

Sets

Sets just store's unique values. The data type to be used is specified inside triangle brackets **<string>.** You will need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using sets in your program.

make a set:

```
        set<string> set1;
```

add values to a set:

```
         set1.insert("Cat");
         set1.insert("Tiger");
         set1.insert("Lion");
         set1.insert("Lion");
```

print out the size of the set:

```
         size = set1.size();
```

```
cout <<  "I have a set of "  << size  << " animals " << endl;
```

```
I have a set of 3 animals
```

C++ does not print out the sets for you, so you need to do it your self using loops. A Set has an **iterator** that can be used to print our set item values. An iterator will access the items stored in a set sequentially one by one. Sets are printed out in alphabetically order.

```
// print out set using iterator
set<string>::iterator  itr3;
for(itr3=set1.begin();itr3!=set1.end();itr3++)
        {
        cout << *(itr3) << " ";
        }
cout << endl;
```

```
Cat Lion Tiger
```

Note we only have 3 animals, sets do not contain duplicates.

Find an item in a set, returns an iterator where item found. If item not found returns the end iterator.

```
itr3 = set1.find("Tiger");
if(itr3 != set1.end())
        cout << "Tiger found" << endl;
else
        cout << "Tiger not found" << endl;
```

remove item by value from set:

```
set1.erase("Tiger");
```

Print out set:

```
set<string>::iterator  itr3;
 for(itr3=set1.begin();itr3!=set1.end();itr3++)
          {
          cout << *(itr3) << " ";
          }
cout << endl
```

```
Cat Lion
```

make another set called set2:

```
set<string> set2;
```

add value's to set:

```
set2.insert("Cat");
set2.insert("Dog");
set2.insert("Lion");
```

Print out set using iterator:

```
for(itr3=set3.begin();itr3!=set3.end();itr3++)
           {
           cout << *(itr3) << " ";
           }
cout << endl;
```

```
Cat Dog Lion
```

make empty set 3:

```
set<string> set3;
```

take intersection of set1 and set2 and print results:

```
set_intersection(set1.begin(),set1.end(),set2.begin(),set2.end(),inserter(set3,set3.begin()));
```

The **inserter** function is used to insert the elements into the result set3.

Print out set using iterator:

```
cout << "intersection" << endl;
 set<string>::iterator  itr3;
 for(itr3=set3.begin();itr3!=set3.end();itr3++)
              {
              cout << *(itr3) << " ";
              }
cout << endl;
```

```
Cat Lion
```

The intersection of 2 sets is a set of the items that are the same in both sets

make set 4:

```
set<string> set4;
```

take union of set1 and set2 and print results:

```
set_union(set1.begin(),set1.end(),set2.begin(),set2.end(), inserter(set4,set4.begin()));
```

The **inserter** function is used to insert the elements into the result set3.
Print out set using iterator:

```
cout << "union" << endl;
for(itr3=set4.begin();itr3!=set4.end();itr3++)
             {
             cout << *(itr3) << " ";
             }
cout << endl;
```

```
Cat Dog Lion
```

The union of 2 sets is a set of the items that are the included in both sets

Both the **set_intersection** and the **set_union** functions also use the **inserter** function that is used to inset the elements in the result set.

To use these funcions you need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using **these** functions in your program.

Printing unique words in a string

Assign a word to a string like "tomorrow".

```
string s = "tomorrow";
```

Make a set and insert each letter into the set.

```
set<char>  set4;

for(unsigned int i=0;i<s.length();i++)
{
        set4.insert(s[i])
}
```

Print out the set, you should print out all the unique letters of the word.

```
set<char>::iterator  itr4;

 for(itr4=set4.begin();itr4!=set4.end();itr4++)
          {
          cout << *(itr3) << " ";
          }
cout << endl;
```

```
m o r t w
```

todo : try more words

Map

A Map contain a **key** and a **value**. A map can have many keys and corresponding values. Think of a map is like a telephone book with the name as the key and the telephone number as the value.

A map needs to know what type of data type for the key and the data type for the value it will be using. The data type is specified inside diamond <> brackets like <string, string>. The first data type is for the key and the second data type is for the value.

You will need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using map in your program.

We then make a empty map as follows:

```
    // make empty map
    map<string, string> map1;
```

Next we add some keys and corresponding values:

```
    // add keys and values to map
    map1["name"] = "Tom";
    map1["age"] = "24";
    map1["idnum"]  = "S1234"
    // get values from a map
    cout <<  map1["name"] << endl; // Tom
    cout << map1["age"] << endl; // 24
    cout << map1["idnum"] << endl; // S1234
```

Alternatively you can use the insert method and a **pair** object to insets name and a value into a map.

```
        map1.insert(pair<string, string>("name", "sue"));
```

A **pair** object contains a name and a value access by first and second data members.

C++ does not print out the map as easy as other programming languages, we must do it ourselves using the map iterator and a loop. The map is print out automatically sorted by key. Print out map using a iterator:

```
          map<string, string>::iterator itr4;
```

The iterator points to a pair object. From the iterator the **key** is printed using the **first** member and the **value** is printed out using the **second** member:

```
           for(itr4 = map1.begin();itr4 != map1.end();itr4++)
              {
              cout << itr4->first << " : " << itr4->second << endl;
              }
        cout << endl;
```

```
age:24
idnum:S1234
name:Tom
```

We print a list of keys in a vector. The keys are already sorted

Print a list of keys from the map

```
 // get list of keys
   vector<string> v2;
   map<string,string>::iterator itr5;
   for(itr5= map1.begin(); itr5 != map1.end(); itr5++)
   {
   v2.push_back(itr5->first);
   cout << itr5->first << endl;
   }
```

```
age
idnum
name
```

Print a list of values from the map

```
vector<string> v3;
map<string,string>::iterator itr6;

for(itr6= map1.begin(); itr6 != map1.end(); ++itr6)
{
  v3.push_back(itr6->second);
  cout << itr6->second << endl;
}
```

```
24
S1234
Tom
```

Print Map Sorted by Value

We will now print out a map sorted by value. We first fill a vector with the map pairs that contain a key and a corresponding value.

```
// put map  key and value pairs in a vector
vector<pair<string, string> > pairs(map1.begin(), map1.end());
```

Next we sort the vector pairs by value. You will need to use
the following comparator to sort the vector pairs by value

You need to put the comparator function at the top of the lesson8.cpp file just before the main function. & means pass by reference which passes an absolute address to the function. Pass by reference is good to use because only the address is passed to the function and the value at that address can be easily changed. Pass by value would only pass a value to the function and the value will only change inside the function but not outside the function.

```
bool cmp_second(const pair<string,string> &a,const pair<string,string> &b)
{
    return a.second<b.second;
}
```

```
// sort vector key value pairs
sort(pairs.begin(), pairs.end(),cmp_second);

// print out key value vector pairs
 vector<pair<string, string> >::iterator itr7;

      for(itr7=pairs.begin();itr7!=pairs.end();itr7++)
                {
                cout << itr7->first << " : " << itr7->second << endl;
                }
   cout << endl;
```

```
age : 24
idnum : S1234
name : Tom
```

To do:

change the > to < in the cmp_second comparator function

Using lambda

It is very inconvenient to add small additional function to a program ever tine you need a piece of code to do something. There is a mechanism that allows you to add a small inline function in your code instantly when you need it. The function does not get a name and is known as an anonymous function, having no specified name its gets a default name called **lambda**. We can now put our comparator function right inside the sort function call.

Our lamda comparator function looks like this:

```
[](const pair<string, string> & a, const pair<string, string> & b) -> bool
{
   return a.second < b.second;
}
```

The function name is: **[]**

The input parameter list is:
(const pair<string, string> & a, const pair<string, string> & b)

The return data type is specified as: **-> bool**

The function code is:

```
{
   return a.second < b.second;
}
```

Which really means:

```
bool [](const pair<string, string> & a, const pair<string, string> & b)
{
   return a.second < b.second;
}
```

(The syntax is just a little different)

Our sort call now looks like this, so much easier, no external sort function is needed, and everything is contained in one programming statement.

```
// sort pairs vector
sort(pairs.begin(), pairs.end(),
   [](const pair<string, string> & a, const pair<string, string> & b) -> bool
{
   return a.second < b.second;
});

// print out vector pairs
 for(itr7=pairs.begin();itr7!=pairs.end();itr7++)
                 {
                 cout << itr7->first << " : " << itr7->second << endl;
                 }
   cout << endl;
```

To check if an element is in the map use the count method.

```
        if(map1.count("name")>0)
        {
            cout << map1["name"] << endl;
        }
```

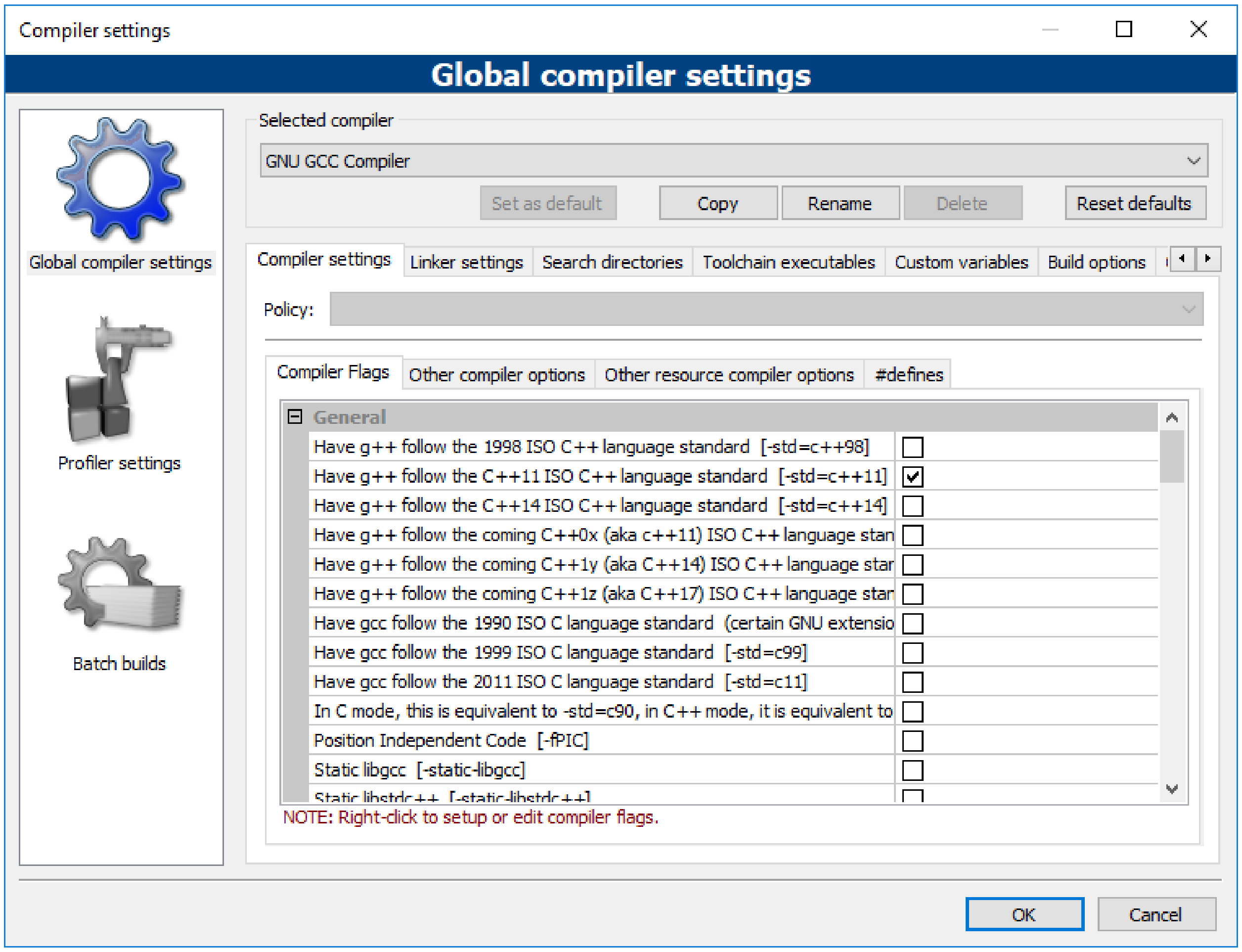
```
Tom
```

To remove an item from the map use the erase method on the key

```
        map1.erase("name");
```

Type all the above examples in your file lesson6.cpp and, make sure you get the same results.

To use lambda functions you need to set your complier to –std=c++11. In code blocks you go to settings menu then select compiler then select c++11 checkbox



Selected compiler: GNU GCC Compiler

Policy:

General

Have g++ follow the 1998 ISO C++ language standard [-std=c++98]	☐
Have g++ follow the C++11 ISO C++ language standard [-std=c++11]	✓
Have g++ follow the C++14 ISO C++ language standard [-std=c++14]	☐
Have g++ follow the coming C++0x (aka c++11) ISO C++ language stan	☐
Have g++ follow the coming C++1y (aka C++14) ISO C++ language star	☐
Have g++ follow the coming C++1z (aka C++17) ISO C++ language star	☐
Have gcc follow the 1990 ISO C language standard (certain GNU extensio	☐
Have gcc follow the 1999 ISO C language standard [-std=c99]	☐
Have gcc follow the 2011 ISO C language standard [-std=c11]	☐
In C mode, this is equivalent to -std=c90, in C++ mode, it is equivalent to	☐
Position Independent Code [-fPIC]	☐
Static libgcc [-static-libgcc]	☐
Static libstdc++ [-static-libstdc++]	☐

NOTE: Right-click to setup or edit compiler flags.

Lesson 9 Homework

Question 1

Make an vector called v1 of your favorite animals like: elephant, cat and dog. Print out the vector of animals.

Ask the user of your program to type in name of one of the animal names from your animals, that they don't like.

Remove the animal from this list and put into another vector called v2.

Then ask them to type in the name of an animal they do like. Add this name to v1 and to v2.

Print out the animals in v1.
Print out the animals in v2.

Put all the code in your homework 9 file.
Call your cpp file Homework9.cpp.

Question 2

Repeat Question 1 but use a list rather than a vector.
Put all the code in your homework 9 file.
Call your cpp file Homework9.cpp.

Question 3

From question1 or question 2 put all the animals from animal list1 into a set called set1.

Then take all the animals from list2 and put into another set called set2.

Print out the common animals between the two sets, then print out all the animals that are in both sets.

Put all the code in your homework 9 file.
Call your cpp file Homework9.cpp.

Question 4

Fill a map called map1 with names and animals. The name should be the key and the animal should be the value.

Example: "fluffy" "cat"

Fill a second map called map2 with animals and sounds,

Example: **"cat" "meow"**

Print out all the key names to the screen. In a loop ask the user to type in one of the names. Lookup up the name in map1 and obtain the animal. Next ask the user what sound the animal makes.

Example:
Select a name: tom, bill, sally, sue
The user would type in sally

The program would respond:
sally is a cat

Next ask the user what sound the animal makes? Use map2 to look up the sound the animal makes. If the user is correct the tell them they are correct, else tell them what sound the animal makes.

Example:
What sound does a cat make?
The user would type in "purr"

The program would say
You are incorrect
cat's meow.

Anytime user types "exit" or a blank the program exits. If they type in a wrong name let them know.

You can use the count method of the map class to check if a key name is in the map.

```
if(map1.count(name)==0)
{
   cout << "name: "  << "not known " <<  endl;
}
```

Put all the code in your homework 9 file.
Call your cpp file Homework9.cpp.

Question 5

Sentence Generator

A Sentence is composed of the following:
<article><adjective><noun><adverb><verb><article><adjective><noun>

Make an vector<string> of articles like: "a", "an" and "the"
Then make an vector <string> of adjectives like: "fat", "big", "small"
Then make an vector <string> of nouns like: "cat", "rat", "house"
Then make an vector <string> of adverbs like: "slowly", "gently", "quickly"
Then make an vector <string> of verbs like "ate", "sat on", "pushed"

EACH LIST SHOULD HAVE THE SAME AMOUNT OF ENTRIES LIKE 3 ENTRIES EACH

Make a map<String, vector <string >> called words to hold all the lists:

words ["articles"] = articles;
words ["adjectives"]=adjectives;
words ["nouns"]=nouns;
 words ["adverbs"]=adverbs;
 words ["verbs"]=verbs;

Next make a vector<string>   of the map keys (parts of speech) to make a sentence:
"articles", "adjectives", "nouns", "adverbs", "verbs", "articles", "adjectives", "nouns"

Finally make a sentence using the map entries, using the key parts of speech vector and by selecting random words from the words map values vector.

You need to first seed the random number generator with the time of day to be able to get random sentences.

```
#include <cstdlib>
#include <ctime>

srand((unsigned int)time(0));
```

Next make a loop to print out the random sentences:

```
string sentence = "";
for(unsigned int i=0;i<keys.size();i++)
{
   int r = rand()%3;
   string key = keys[i];
   sentence += words[key][r] + " ";
}
```

Then print out the sentence:

```
cout << sentence << " ." <<endl;
```

You should get something like this:

```
The big cat slowly ate the small rat
```

Which has picked random words from the dictionary sentence structure:
<article><adjective><noun><adverb><verb><article><adjective><noun>

Put all the code in your homework 9 file.
Call your cpp file Homework9.cpp.

LESSON 10 File Access

For his lesson make a new C++ source file called Lesson9.cpp and in the main function for each section type in the following programming statements.

File Access

C++ has extensive file objects for reading and writing to file. We concentrate on the most used. You will need to use the following includes for file access

```
#include <fstream>
#include <cstdlib>
```

We will use the following input test file. Just make an empty text file and type in the following contents.

Write character to a file

We use the **putc** method of the ofstream class to write characters one by one sequentially to a file. As before we open the file then check if the file is open and the write characters to the file. If you do not close the file, then the contents of the file will be lost.

```
// write character to a file

    // open file
    ofstream fout("test1.txt");

     // check if file opened
    if (!fout)
      {
     // report cannot open file
     cout << "cannot open file: test1.txt" << endl;
     exit(1);
      }
```

```
// characters to write to file
string s = "Hello";

// write characters to file
for (unsigned int i = 0; i < s.length(); i++)
{
   char c = s[i];
   cout << c;
   fout.put(c);
}

cout << endl;
fout.put('\n');
fout.close();
```

test1.txt

```
Hello
```

Read characters from a file

The **ifstream** class is used to read characters one by one sequentially from a file. The **get** method from the **ifstream** class is used to read characters from a file. Each char from the file is read as an int so that the end of file EOF indicator -1 can be acknowledged. The constructor of the ifstream class open a file using the input file name. We check if a file is open by using the !operator on the ifstream object. If the file cannot be opened, we terminate the program using the exit function. After all characters have been read from the file we close the file using the close method.

```
// read character from a file

    // open file
    ifstream fin("test1.txt");

    // check if file opened
    if (!fin)
    {
       // report cannot open file
       cout << "cannot open file: test1.txt" << endl;
       exit(1);
    }

      // get first char in file
      int ch = fin.get();

      // loop to end of file
      while (ch != -1)
         {
            cout << (char)ch; // print out char
            ch = fin.get(); // get next char
         }

         fin.close(); // close file
      }
```

```
Hello
```

write lines to a file

The **ofstream** class is also used to write lines one by one sequentially to a file. We just use the << operator from our fout object. Since we have already declared an ofstream object we need to use the open method of the ofstream object instead to open a file. We first clear the ofstream object before opening it to reset the ofstream object.

```
// write lines to a file

 // open file
 fout.clear();
 fout.open("test2.txt");

  // check if file opened
 if (!fout)
 {
  // report cannot open file
  cout << "cannot open file: test2.txt" << endl;
  exit(1);
  }

    // write lines to file
    fout << "Hello there"  << endl;
    fout << "I like programming << endl;

    fout.close();
```

test2.txt

```
Hello there
I like programming
```

Read line by line from a file

To read lines from a file line by line we use the getline function that takes in a ifstream object and a string object. Since we have already declared an ifstream object we need to use the open method of the ifstream object to open a file. We first clear the ifstream object before opening the file to reset the ifstream object.

We keep reading lines to the end of file is found. When the fin object returns false the end of file has been reached. Some Unix compilers work different from Windows compilers. The getline function may retain the end of line terminator \n. We use the find and erase methods of the string class to remove the \n.
After we read all the lines from the file the file must be closed.

```
// read lines from a file

   // open file
   fin.clear();
   fin.open("test2.txt");

   // check if file opened
   if (!fin)
   {
      // report cannot open file
      cout << "cannot open file: test2.txt" << endl;
      exit(1);
   }

  // read first line from file
     string line;
     getline(fin,line);
```

```
          // loop to end of file
          while (fin)
             {
                // remove \n
                if(line.find('\n') != string::npos)
                   line.erase(line.find('\n'));
                cout << line << endl; // print out line
                 getline(fin,line); // get next line
             }

      fin.close(); // close file
   }
```

```
Hello there
I like programming
```

Read words from a file

Using the ifstream object we can read word by word from a file. The fin stream object reads word by words separated by spaces.
After we read all the words from the file the file must be closed.

```
    // read words from a file

       // open file
       fin.clear();
       fin.open("test2.txt");

       // check if file opened
       if (!fin)
       {
          // report cannot open file
          cout << "cannot open file: test2.txt" << endl;
          exit(1);
       }
```

```
        // read first word from file
           string word;
           fin >> word;

           // loop to end of file
           while (fin)
             {
                cout << word << endl; // print out word
                fin >> word; // get next word
             }

      fin.close(); // close file
    }
```

```
Hello
 there
I
like
programming
```

Append line to end of file

We can also write lines to the end of a file (append) using the ofstream class, The second object of the constructor or open method specifies fie open modes. We use the directive ios::app which states to open file in append mode

```
        // append lines to a file

        // open file
        fout.clear();
         fout.open("test2.txt",ios::app);

         // check if file opened
         if (!fout)
         {
            // report cannot open file
            cout << "cannot open file: test2.txt" << endl;
            exit(1);

         }
```

```
// characters to write to file
s = "Hello";

  // write 3 hello lines to file  test2.txt
  fout << s << endl;
  fout << s << endl;
  fout << s << endl;

  fout.close();  // close file
```

```
Hello
Hello
Hello
```

write lines to a csv file

A csv file is a file where data are stored row by row in columns separated by commas. The **ofstream** class is also used to write lines one by one sequentially to a file. We just use the << operator from our fout object. Since we have already declared an ofstream object we need to use the open method of the ofstream object instead to open a file. We first clear the ofstream object before opening it to reset the ofstream object.

```
// write lines to a file

 // open file
 fout.clear();
 fout.open("test.csv");

  // check if file opened
  if (!fout)
  {
  // report cannot open file
  cout << "cannot open file: test3.csv" << endl;
  exit(1);
   }
 // write csv lines to file
fout << "one,two,three,four"<< endl;

 fout.close(); // close file
```

```
one,two,three,four
```

Read a csv file.

A csv file is a file where data are stored row by row in columns separated by commas. The **ifstream** class is used again to read lines one by one sequentially from the file. In a for loop we use istringstream class to scan a line using the getline function that will find each comma and then put the data for that row and column in a vector object to be used later. The data retrieved are also called **tokens**. We also use the **istringstream** class to retrieve each word from the line separated by commas. We use the **getline** method with the **istringstream object** And specify the delimeter as ',' to read each word from the line.

To use the **istringstream** class you need to

```
#include <sstream>
```

At the top of your cpp or h file,

File: test.csv

```
one,two,three,four
```

```
// read lines from a csv file

   // open file
   fin.clear();
   fin.open("input.csv");
```

```
  // check if file opened
  if (!fin)
  {
     // report cannot open file
     cout << "cannot open file: test.csv" << endl;
     exit(1);
  }

// read first line from file
   getline(fin,line);

   // loop to end of file
   while (fin)
      {
         // remove  '\n'
         if(line.find('\n') != string::npos)
               line.erase(line.find('\n'));

         // split lines into tokens
         istringstream sin(line);
         vector<string>v;
         string word;
         getline(sin,word,',');

        // store words in a vector
       while(sin)
        {
        v.push_back(word);
        getline(sin,word,',');
        }

       // print out vector
      for(int i=0;i<v.size();i++)
      {
     cout << v[i]<< endl;
     }
```

```
        getline(fin,line); // get next line
        }

    fin.close(); // close file
    }
```

The tokens are also stored in the vector tokens.
Output token words:

```
one
two
three
four
```

Writing and Reading Records to and from a file

Records are the data variable values defined in a class written to a binary file. A binary file differs from a text file since it stores binary values where as a text file only contains printable values. The values may be the same it is just the way they are interpreted. For example hex value 10 is interpreted as a new line in a text file but in a binary file it is just the value 10.

To write to a binary file you need some data record. A record can be a class or a structure. The data variables declared in a structure or class must be fixed lengths, therefore the **string** object cannot be used. In this situation we use a character string instead. For our example we will use the Book class as follows:

```
class Book
{
private:
  char ISBN[20];
  char title[50];
  double price;
public:

  // construct a book
   Book(string isbn="", string t="", double p=0)
   {
      strcpy(ISBN,isbn.c_str());
      strcpy(title,t.c_str());
      price = p;
   }

   // display a book
  friend ostream& operator <<(ostream& out, const Book& b)
  {
   out << b.ISBN << " " << b.title << " $" << b.price << endl;
   return out;
  }
};
```

Notice we are using old fashion character strings arrays for ISBN and title. We are still using string objects for convenience in our constructor parameters. We use the **cstring strcpy** functions to copy the contents of a string object to a character string. The string object has the method **c_str()** to convert a string object to a character array.

```
strcpy(ISBN,isbn.c_str());
```

To use the strcpy funcions you need to place

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

at the top of your .cpp or .h file so that the compiler will know that you are using the **strcpy** function in your program.

The first thing we need to do is write some book records to a file. Each record is the data variable values defined in the Book class.

We first make a book object

```
Book book1("123456789","Happy Days",23.56);
```

Then write the book record to the file using the **write** method of the **ifstream** class. The **sizeof** method calculates the total number of data bytes in the Book object.

```
fout.write((char*)&book1,sizeof(Book));
```

When we open the file we use the **ios::binary** flag to set the file to binary mode.

```
// write records to a file

// open file
 fout.clear();
 fout.open("records.bin",ios::binary);


// check if file opened
if (!fout)
{
  // report cannot open file
  cout << "cannot open file: records.bin" << endl;
  exit(1);
}

  // write book to file
  Book book1("123456789","Happy Days",23.56);
  fout.write((char*)&book1,sizeof)Book));

  Book book2("876543245","Wizard of Oz",19.96);
  fout.write((char*)&book2,sizeof(Book));
  fout.close();
```

Once we write some book records to the file we can read back the records and display them on the console screen. We use the **read** method from the **ofstream** class to read book records stored previously on the file.

```
 // read from binary file

// open file
   fin.clear();
   fin.open("records.bin",ios::binary);

 // check if file opened
 if (!fin)
 {
    // report cannot open file
    cout << "cannot open file: records.bin" << endl;
    exit(1);
 }

 // read records from a file
 Book book;

 while(fin.read((char*)&book,sizeof(Book)))
     {
         cout << book << endl;
     }

      fin.close();
```

```
123456789 Happy Days $23.56

876543245 Wizard of Oz $19.96
```

We can also add new records to the end of the file using the **ios::app flag.**

```
 // append records to a binary file

   // open file
   fout.clear();
   fout.open("records.bin",ios::binary|ios::app);
 // check if file opened
   if (!fout)
   {
      // report cannot open file
      cout << "cannot open file: records.bin" << endl;
      exit(1);
   }

      // write book to file
      Book book3("87654542","Alice in Wonderland",18.88);
      fout.write((char*)&book3,sizeof(Book));
      fout.close();
```

Again we read the book records from the binary file and display on the console screen.

```
   // read from binary file

  // open file
     fin.clear();
     fin.open("records.bin",ios::binary);

   // check if file opened
   if (!fin)
   {
      // report cannot open file
      cout << "cannot open file: records.bin" << endl;
      exit(1);
   }

   // read records from a file
   while(fin.read((char*)&book,sizeof(Book)))
       {
           cout << book << endl;
       }

        fin.close();
```

```
123456789 Happy Days $23.56

876543245 Wizard of Oz $19.96

87654542 Alice in Wonderland $18.88
```

Open binary file for simultaneously Read and Write

Opening a file for reading and writing is very convenient, in this case we use an **fstream** object and use the flags **fin.clear()** and an **ios::out** to open a file for input and output at the same time**.** We also use the **ios::ate** flag so we that the first record can be added to the end of the file.

```
   // open file for read/write
   fstream fio("records.bin",ios::binary|ios::in|ios::out|ios::ate);

  // check if file opened
  if (!fio)
  {
     // report cannot open file
     cout << "cannot open file: records.bin for read and write" << endl;
     exit(1);
  }

     // write book to file
     Book book4("765344532","Open Skies",12.78);
     fio.write((char*)&book4,sizeof(Book));
```

Once we write a new record to the end of the file we can go the start of the file and read each record one by one and display on the console screen.
We use the **seekg(position)** method from the **fstream** class to set the file pointer to the start of the file.

Example: **fio.seekg(0)** will set the file pointer to position 0.

```
  // read from binary file
  fio.seekg(0);

  while(fio.read((char*)&book,sizeof(Book)))
     {
        cout << book << endl;
     }
```

```
123456789 Happy Days $23.56

876543245 Wizard of Oz $19.96

87654542 Alice in Wonderland $18.88

765344532 Open Skies $12.78
```

We can read from any position on the file using **seekg(position)** and write to any position using **seekp(position).** We now write a new book to record position 2. The formula is:
file record position = record number * size of record

Record position and record numbers start at 0. Note we first clear the fstream object before we set the record position, this is because when we read all the records from the file the end of file flag was set.

```
fio.clear();
fio.seekp(2*sizeof(Book));

  // write book to file
   Book book5("3443223475","Hello World",6.89);
   fio.write((char*)&book5,sizeof(Book));
```

We read all records again:

```
fio.seekg(0);

while(fio.read((char*)&book,sizeof(Book)))
    {
       cout << book << endl;
    }
```

```
123456789 Happy Days $23.56

876543245 Wizard of Oz $19.96

3443223475 Hello World $6.89

765344532 Open Skies $12.78
```

We can specify which record to read using **seekg(position).** Here we read record 2 from the file.

```
fio.clear();
fio.seekg(2 * sizeof(Book));

fio.read((char*)&book,sizeof(Book));

cout << book << endl;
```

```
87654542 Alice in Wonderland $18.88
```

Always close the file when you are finished using it or you will lose all your data.

```
fio.close();
```

Lesson10 Homework To do

Question 1

Open a text file for write and write a small 5 line story in it then close the file. Next open the file for read and count the number of letters, word, sentences and lines. Words are separated by spaces and new lines. Sentences are separated by periods "." or other punctuation like "?".
 Lines are separated by '\n'. Words may contain numbers and punctuation like apple80 and hyphens like don't.

Print a report to the screen: the number of letters, words, sentences and lines. Also write the report to a file called report.txt. Open the report file and display the report file lines to the screen. Call your cpp file Homework10.cpp.

Question 2

Write a program that writes out another C++ program to a file. Then open up the file you wrote that contains the C++ program and execute it.

Algorithm:

1. Open up a file for write with a cpp extension like “test.cpp”
2. Write lines to a file with print statements like:
3. fout << "#include <iostream> " << endl;
3. fout << "using namespace std;" << endl;
3. fout << "int main()" << endl;
3. fout << "{" << endl;
3. fout << “fout << \"I like Programming\" << end; " << endl;
   you need to use "\"" as additional double quotes
3. fout << "return 0"<< endl;
3. fout << "}" << endl;

4. close the file
5. open the test.cpp file in your C++ IDE and run the program
   It should print out

```
I like programming
```

Call your C++ program Homework10_2.cpp.

Lesson 11 VIRTUAL METHODS, ABSTRACT CLASSES and POLYPHORISM

Virtual Methods

A virtual method is executed in the derived class not in the base class when the object data type is a base class. This is known as **overriding**. A virtual method starts with the keyword **virtual**.

virtual return_data_type method_name(parameter_list)

For an example the following Employee class has the virtual method toString.

```
virtual string toString();
```

Note: **overriding** means same method name and identical parameter data types. Do not confuse with **overloading** where we have same method name but different parameter data types.

Here is the Employee class with the virtual toString method:

```
class Employee
{

private:
   string name;
   string id;
   double salary; // yearly salary

public:

   // construct Employee with name, id and salary
   Employee(string name,string id,double salary);

   // return employee info
   virtual string toString();  // <<<<<<<<<<<======= virtual method
};
```

A derived class of the Employee class like the Manager class would have the overridden toString(). The overridden toString method of the Manager class is not virtual.

```
string toString();
```

Here is the Manager derived class having the overridden toString() method:

```
#include "Employee.h"

class Manager:public Employee
{
public:

// construct Manager with name, id and salary
   Manager(string name,string id,double salary);

   // return manager info
   string toString();  // <<<<<<<<<<<========  overridden toString method
};
```

Note: the derived class does not have the virtual keyword.

When a manager is instantiated the toString method is called from the Manager class rather than from the Employee class because the toString method in the Employee class is marked virtual.

```
Employee* emp = new Manager("Tom Smith","M1234",100000);
```

Even though the pointer data type is an **Employee*,** the toString() method of the manager class is called rather than the toString method from the Employee class. This is known as **overriding** . The toString method of the Manager class overrides the toString method of the Employee class. If the Employee class toString method did not have the **virtual** keyword then the toString method of the employee class would be called instead.

If the pointer data type was a Manager* then the toString method of the Manager class would be called regardless if the Employee toString method was virtual or not virtual. This is because the data type of the pointer is a Manager.

```
Manager* emp = new Manager("Tom Smith","M1234",100000);
```

Derive classes are independent if their base class. Note: All virtual methods declared in a class must have implemented code. The derive class does not need to implement all virtual methods in the base class.

Todo:

Make an Employee class and a Manager class. Instantiate the Manager class in two different ways. One way with an Employee pointer and the other way with an Manager pointer. In both ways make the Employee toString virtual and non virtual. You will have 4 different cases to try out.

Abstract Classes

Abstract classes are base classes that have methods to be implemented by its derived class. These methods are called abstract methods and have no code in the base class. Classes that have abstract methods are known as Abstract classes. The code in the abstract class is to be implemented in the derived classes. Abstract classes can never be instantiated since the class contains incomplete code in some methods. Note: Non Abstract classes are called **Concrete** classes.

In our Employee class we can make abstract method to calculate the employee's wages for the week. A abstract method is a method, that is defined in the base super class that has a **virtual** method header definition but no programming statements and is equated to 0. Abstract methods in C++ are known as **pure virtual** methods.

```
virtual double weeklyPay()=0; // pure virtual method
```

The programming statements for a **pure virtual method** will be defined in the derived class not in the super base class.

Virtual methods without = 0 are just virtual methods that must be implemented in the super base class but may be optionally overridden by the derived class.

```
virtual string toString();  // virtual method
```

virtual methods without = 0 are not abstract methods because the code must be implemented in the class they are declared in. they are just known as a virtual method. virtual methods with = 0 are abstract methods because the code must be implemented in the derived class and are known as a pure virtual method.

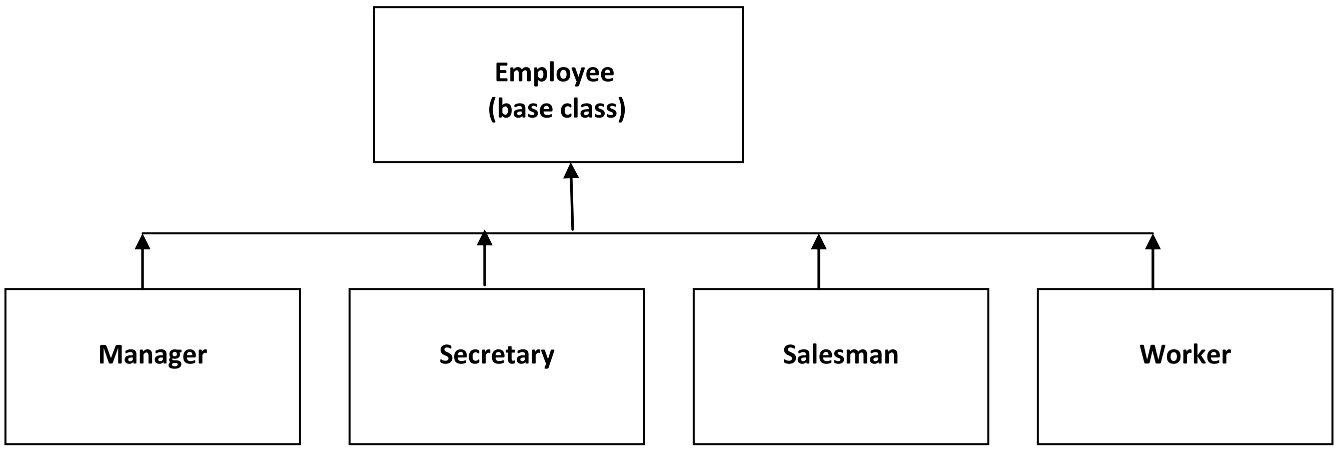
```
virtual double weeklyPay()=0; // pure virtual method
```

The weeklyPay method in the Manager class may be implemented as follows:

```
// calculate weekly pay
double Manager::weeklyPay()
{
   double pay = getSalary() / WEEKS_IN_YEAR;
   return pay;
}
```

Polymorphism

Polymorphism is another powerful concept in Object Oriented Programming. Polymorphisms allows a super base class to represent many other different derived classes. Polymorphisms executer's a method that has the same name from each derived class but produces a different behaviour. To demonstrate Polymorphism, we will have an Employee super base class to represent many kinds of Employees: Managers, Secretaries, Salesman and Workers derived classes.



Make a new C++ source file called Employee.cpp and a C++ header file called Employee.h Your will need private variables: name, employee ID number and salary. Make constructers, getters, setters and toString methods.

Here is our Employee abstract class:

```
/*
 * Employee.h abstract class
 *
#ifndef EMPLOYEE_H_INCLUDED
#define EMPLOYEE_H_INCLUDED

#include <iostream>
#include <iomanip>
#include <sstream>
#include <string>

using namespace std;
```

```
class Employee
{
public:
   const int WEEKS_IN_YEAR = 52;
   const int HOURS_IN_WEEK = 40;

private:
   string name;
   string id;
   double salary; // yearly salary

public:

   // construct Employee with name, id and salary
   Employee(string name,string id,double salary);

   // return employee name
   string getName();

   // assign employee name
   void setName(string name);

   // return employee id
   string getID();
   // assign employee id
   void setID(string id);

   // return employee yearly salary
   double getSalary();

   // assign employee yearly salary
   void setSalary(double salary);

   // calculate weekly pay
   virtual double weeklyPay()=0;
   // return employee info
   virtual string toString();
};

#endif // EMPLOYEE_H_INCLUDED
```

Here is the Employee.cpp implementation file:

```
/*
 * Employee.cpp
 *
 */

#include "Employee.h"

         // construct Employee
         Employee::Employee(string name,string id,double salary)
         {
         this->name=name;
         this->id=id;
         this->salary = salary;
          }

         // return employee name
         string Employee::getName()
         {
            return this->name;
         }

         // assign employee name
         void Employee::setName(string name)
         {
            this->name = name;
         }
```

```
// return employee id
string Employee::getID()
{
   return this->id;
}

// assign employee id
void Employee::setID(string id)
{
   this->id = id;
}

// return employee yearly salary
double Employee::getSalary()
{
   return salary;
}

// assign employee yearly salary
void Employee::setSalary(double salary)
{
   this->salary = salary;
}

// return employee info
string Employee::toString()
{
   ostringstream sout;
   sout << name << " " << id << " $" << fixed << setprecision(2) << salary;
   return sout.str();
}
```

We have done some formatting in our toString method, we have used format manipulators from

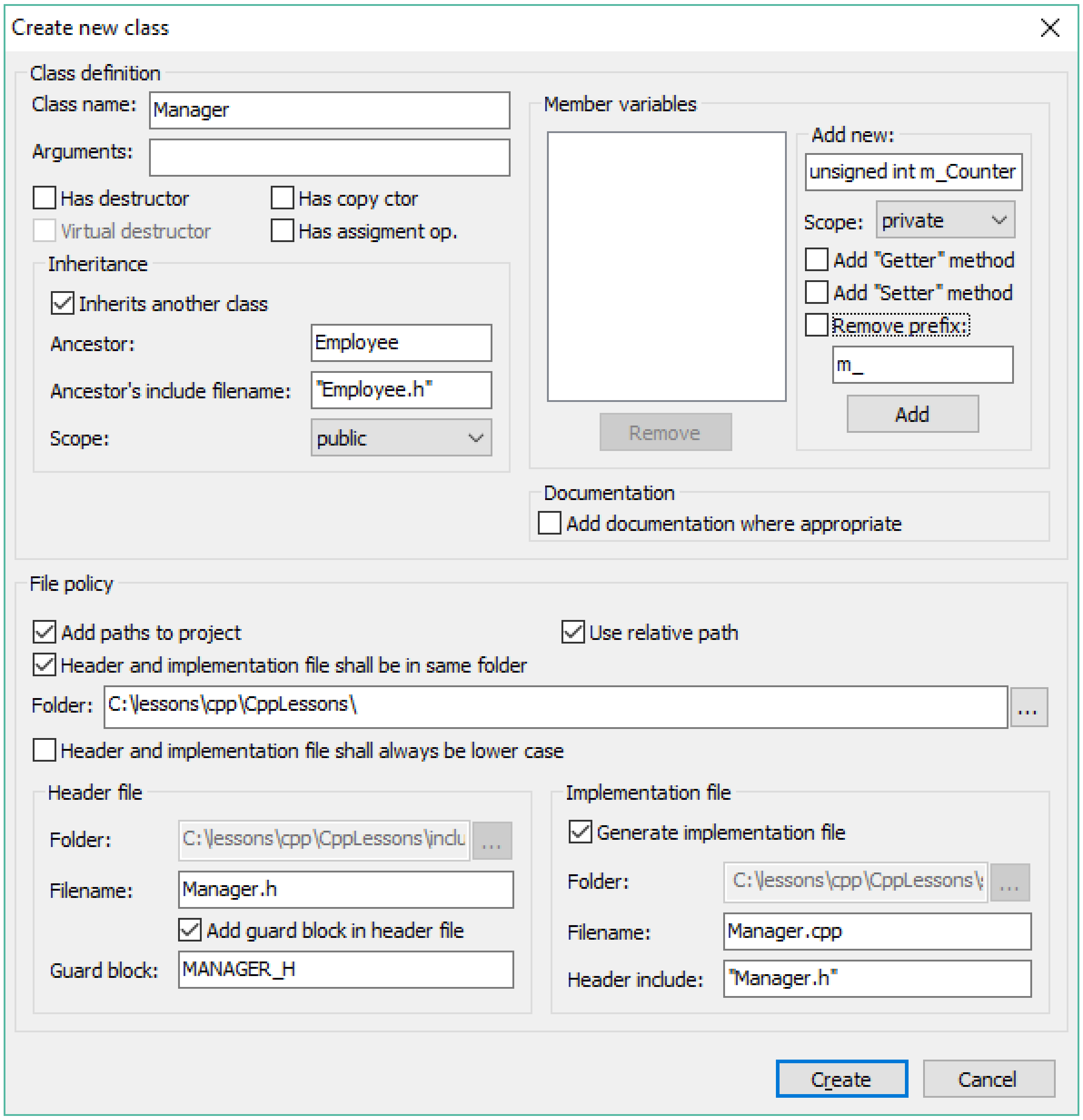
```
#include  <iomanip>
```

manipulator	description	example
Fixed	Use decimal point number	Fixed
setprecision(n)	Specify number of decimal points	setprecision(2)
setw(n)	Set width	setw(10)
left	Align left	left
right	Align right	right

We now need to make the derived classes. We will have 4 derived classes. Each derived class will calculate the pay for the week differently, calculated from the yearly salary. Each derived class will calculate the weekly pay separately as follows:

Derived class	**How to calculate weekly pay**
Manager	Divide yearly salary by number of weeks in year
Secretary	Divide yearly salary by number of weeks in year plus $100 bonus
Salesman	Divide yearly salary by number of weeks in year plus sales Commission rate
Worker	Divide yearly salary by number of weeks plus any overtime time and a half

We put all the derived class definitions in their separate files. You can make separate header and class files or from the File menu you can select class. This will make both header and implementation files at the same time. Set all the check boxes to the following before pressing the create button.



Create new class

Class definition
Class name: Manager
Arguments:
☐ Has destructor ☐ Has copy ctor
☐ Virtual destructor ☐ Has assigment op.

Inheritance
☑ Inherits another class
Ancestor: Employee
Ancestor's include filename: "Employee.h"
Scope: public

Member variables
Add new: unsigned int m_Counter
Scope: private
☐ Add "Getter" method
☐ Add "Setter" method
☐ Remove prefix: m_

Documentation
☐ Add documentation where appropriate

File policy
☑ Add paths to project ☑ Use relative path
☑ Header and implementation file shall be in same folder
Folder: C:\lessons\cpp\CppLessons\
☐ Header and implementation file shall always be lower case

Header file
Folder: C:\lessons\cpp\CppLessons\inclu
Filename: Manager.h
☑ Add guard block in header file
Guard block: MANAGER_H

Implementation file
☑ Generate implementation file
Folder: C:\lessons\cpp\CppLessons\
Filename: Manager.cpp
Header include: "Manager.h"

Here are the derived .h and .cpp files:

```
/*
 * Manager.h derived class
 */

#ifndef MANAGER_H
#define MANAGER_H

#include "Employee.h"
```

```
class Manager:public Employee
{
public:

// construct Manager with name, id and salary
   Manager(string name,string id,double salary);

   // calculate weekly pay
   double weeklyPay();

   // return manager info
   string toString();
};

#endif // MANAGER_H_INCLUDED

/*
 * Manager.cpp derived class
 *
 */

#include "Manager.h"

// construct Manager with name, id and salary
Manager::Manager(string name,string id,double salary)
             :Employee(name, id,salary)
{
}

// calculate weekly pay
double Manager::weeklyPay()
{
   double pay = getSalary() / WEEKS_IN_YEAR;
   return pay;
}
```

```
// return manager info
string Manager::toString()
{
   return "Manager " + Employee::toString();
}

/*
 * Secretary.h derived class
 */

#ifndef SECRETARY_H
#define SECRETARY_H

#include "Employee.h"

class Secretary : public Employee
{
   // Constants
public:

   const double BONUS = 100;

public:
   // construct Secretary with name, id and salary
   Secretary(string name,string id,double salary);

   // calculate weekly pay
   double weeklyPay();

   // return manager info
   string toString();

};

#endif // SECRETARY_H
```

```
/*
 * Secretary.cpp derived class
 *
 */

#include "Secretary.h"

// construct Secretary with name, id and salary
Secretary::Secretary(string name,string id,double salary)
   : Employee(name, id, salary)
{
}

// calculate weekly pay
double Secretary::weeklyPay()
{
   double pay = getSalary()/ WEEKS_IN_YEAR + BONUS;
   return pay;
}

// return secretary info
string Secretary::toString()
{
   return "Secretary " + Employee::toString();
}

/*
 * Salesman.h derived class
 */

#ifndef SALESMAN_H
#define SALESMAN_H

#include "Employee.h"
```

```
class Salesman: public Employee
{
   // Constants
public:
   static const double COMMISSION_RATE = .25;

private:
   // weekly sales
   double sales;

public:

   // construct Salesman with name, id and salary
   Salesman(string name,string id,double salary, double sales);

   // calculate weekly pay
   double weeklyPay();
   // return salesman info
   string toString();
};

#endif // SALESMAN_H

/*
 * Salesman.cpp derived class
 *
 */

#include "Salesman.h"

// construct Salesman with name, id and salary
Salesman::Salesman(string name,string id,double salary, double sales)
   :Employee(name, id,salary)
{
   this->sales = sales;
}
// calculate weekly pay
```

```
double Salesman::weeklyPay()
{
   double pay = getSalary()/WEEKS_IN_YEAR + sales * COMMISSION_RATE;
   return pay;
}

// return manager info
string Salesman::toString()
{
   ostringstream sout;

   sout << "Salesman " << Employee::toString() << " Sales: $" << fixed <<
   setprecision(2) << sales;
   return sout.str();
}

/*
 * Worker.h derived class
 */

#ifndef WORKER_H
#define WORKER_H

#include "Employee.h"

class Worker: public Employee
{
public:

   // Constants
   const double OVERTIME_RATE = 1.5;

private:

   // hours overtime
   int overtime;

public:
  // construct Worker with name, id and salary
  Worker(string name,string id,double salary, int overtime);

  // calculate weekly pay
  double weeklyPay();

  // return manager info
  string toString();

};
#endif // WORKER_H

/*
 * Worker.cpp derived class
 *
 */

#include "Worker.h"

// construct Worker with name, id and salary
Worker::Worker(string name,string id,double salary, int overtime)
  :Employee(name, id,salary)
{
   this->overtime = overtime;
}

// calculate weekly pay
double Worker::weeklyPay()
{
  double pay_rate = getSalary()/ WEEKS_IN_YEAR / HOURS_IN_WEEK;
  double pay = getSalary()/ WEEKS_IN_YEAR
         + overtime * pay_rate * OVERTIME_RATE;
  return pay;
}
```

```
// return worker info
string Worker::toString()
{
   return "Worker " + Employee::toString();
}
```

Polymorphism is like a giant if-else statement to execute the selected derived object method. For example: If it is a Salesman derived class object, then we calculate weekly wage using sales and commission.

Make a cpp file called Lesson8.cpp and put the following in the main function .Our first step is to make an array to hold Employee derived objects.

```
// number of employees
const int NUM_EMPLOYEES = 4;

// make an array of employees pointers
Employee*  employees[NUM_EMPLOYEES];
```

Next, we will the add the derived objects to the Array. Each derived object gets a name, employee id, yearly salary, the salesman gets the sales for the week and the worker get the number of over time hours for the week.

```
// fill array with derived objects
employees[0] = new Manager("Tom Smith","M1234",100000);
employees[1] = new Secretary("Mary Smith","S5678",40000);
employees[2] = new Salesman("Bob Smith","SM1111",20000,10000);
employees[3] = new Worker("Joe Smith","W2222",30000,5);
```

Next, we loop through the array printing out the employee info and the calculated weekly pay. Notice the weekly pay is different for each employee type, this is what we want, automatic selection. This is polymorphism in action.

```
// loop through employee array
// print out employee info
// calculate weekly pay.
```

```
for (int i = 0; i < NUM_EMPLOYEES; i++)
{
   // print out employee info
   cout << employees[i]->toString() << endl;

   // calculate weekly pay
   double pay = employees[i]->weeklyPay();

   // print weekly pay
   cout << "My weekly pay is: $" << fixed << setprecision(2) << pay << endl;
}
```

Here is the polymorphism output:

```
Manager Tom Smith E1234 $100000.00
My weekly pay is: $1923.08
Secretary Mary Smith E5678 $40000.00
My weekly pay is: $869.23
Salesman Bob Smith E1111 $20000.00 Sales: $10000.00
My weekly pay is: $2884.62
Worker Joe Smith E2222 $30000.00
My weekly pay is: $685.10
```

Todo

Copy all the h and cpp code and put into separate files, and put the main function in a cpp file called lesson8.cpp and then compile all files and run the main function. Make sure you get the same output before proceeding.

Lesson11 Home work

Question 1

In the Employee class make another **pure virtual** method :

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

This will be used to give each employee a yearly raise.

The raise method will be implemented in each derived class. The raise will be calculated from the yearly salary. It could be calculated from percentage of yearly salary or how many hours they work over time. For sales person it could be calculated from their commission. It is up to you how you calculate the raise, but the raise must all be a different, a unique dynamic calculation for each employee. Some employees may not get a raise.

In the main function you will want to calculate the raise after you calculate the weekly pay and add the raise to the weekly pay amount. The weekly raise amount would be a percentage calculated from the yearly raise.

Your main function may now look like this:

```
// print out employee info
 cout << employees[i]->toString() << endl;

 // calculate weekly pay
 double pay = employees[i]->weeklyPay();

// calculate raise
double raise = employees[i]->raise();

// add raise to pay
pay = pay + raise/52;

 // print weekly pay
 cout << "My weekly pay is: $" << fixed << setprecision(2) << pay << endl;

// tell employee about the raise
cout <<  "Which includes a weekly raise of  $" << raise/52 << endl
```

You may update your existing Employee file's and the main function in the lesson11.cpp file rather than making new files. In the real world programs are always being added to.

LESSON 12 INTERFACES and TEMPLATES

An interface is pure abstract class meaning it just has pure abstract methods. Abstract methods are just method declarations but no code. The code is to be implemented in the derived class. The purpose of an interface is to specify what methods a class should have and to represent derived classes that implement the interface. The interface becomes the super base class for the class it implements. An interface can represent many other classes that implement the interface. C++ does not have interface keyword, so we make our own interface using the **class** keyword. All methods in the interface must be **pure virtual**:

virtual method_definition_header = 0;

Pure virtual meaning all method code must be implemented in a derived class but not in the interface itself.

An interface is defined as follows:

```
class interface_name
    {
    virtual method_definition_header(s) = 0;
    }
```

Here is an **ICalculator** interface that defines methods that can be used in a Calculator.

```
/*
 *  ICalculator.h interface
 */

#ifndef ICALCULATOR_H
#define ICALCULATOR_H
```

```
class ICalculator
{
   public:
   virtual double add(double a, double b)=0;
   virtual double sub(double a, double b)=0;
   virtual double mul(double a, double b)=0;
   virtual double div(double a, double b)=0;
};
#endif // ICALCULATOR_H
```

You can put the **ICalculator** interface into a file called ICalculator.h

Here is a **Calculator** class that implements the **ICalculator** interface.

```
class Calculator : public  ICalculator
```

The **ICalculator** is the base class, the **Calculator** class implements the **ICalculator** interface class**.**

```
/* Calculator.h
 * implements Calculator interface
 */
#ifndef CALCULATOR_H
#define CALCULATOR_H

#include "ICalculator.h"

class Calculator : public  ICalculator
{
public:
   double add(double a, double b);
   double sub(double a, double b);
   double mul(double a, double b);
   double div(double a, double b);

};
#endif
```

You can put the Calculator definition class into a file called Calculator.h

Here is the **Calculator** implementation file

```
/*
 * Calculator.cpp
 * implement ICalculator interface
 */

#include "Calculator.h"

double Calculator::add(double a, double b)
{
   return a + b;
}

double Calculator::sub(double a, double b)
{
   return a - b;
}

double Calculator::mul(double a, double b)
{
   return a * b;
}
double Calculator::div(double a, double b)
{
   return a/b;
}
```

You can put the **Calculator** implementation class into a file called Calculator.cpp

TODO

Put the **ICalculator** interface in a file called ICalculator.h file and the **Calculator** class definition in a file called Calculator.h and the Calculator class implementation in a file called Calculator.cpp. In your Lesson12.cpp main function, test all the calculator functions.

You can test like this:

```
Calculator calc;

cout <<  calc.add(3,4) << endl;
```

Alternatively you can make a Calculator test file called TestCalculator.cpp for your main.

Lesson12 Homework

Question 1

Instantiate a Calculator class that implements the ICalculator Interface for **doubles**. In a loop asking the user to type in 2 numbers and what operation they want: + - * / and display the results. You can use 'x' to exit loop.
Put your code in a file called Homework12.cpp

Question 2

Add a accumulator instance variable to the Calculator that will sum up all the results of the calculator. Have methods to retrieve and clear the calculator accumulator. In your main methods do some calculations print out the accumulator. After all calculations have been made clear the accumulator and the print out its value. You need to make a Constructor that initializes the accumulator to 0;
Put your code in a file called Homework12.cpp

TEMPLATES

Templates allow you to specify the data type **implicitly** for a function or **explicitly** for a class that it will use when it runs. Implicit meaning the compiler will figure out the data type for you before the program runs. Explicitly meaning you must specify the data type before the program runs. Templates are quite powerful and impressive, allowing you the convenience to specify different data types for the same function or class definition. C++ has many mechanisms to assist you in your programming adventures.

Function templates

Function templates let you use a function with any data type. For example, you have a function that adds 2 int's like this:

```
int add(int a, int b)
{
return a + b;
}
```

but you want to add 2 doubles instead.

To do this we change our add function to a **template function**. A template function starts with key word **template <typename T>** where T is the unknown data type to be substituted, with the data type you want to use.

Templates can also be specified as **template <class T>** which as nothing to do with a class but sometimes means the data type T will represent a class object instead. They both almost work the same and either one can be used.
Here is our template add function:

```
template <typename T>
T add(T a, T b)
{
   return a + b;
}
```

You use template function just like a regular function.
 "Only the compiler knows for sure".

Try it out:

Put the template function above the top of your main in your Lesson9.cpp file. Then at the bottom of your main type in the following:

```
    // add doubles
    cout <<  add(4.6, 7.8) << endl;

    // add ints
    cout <<  add(4, 7) << endl;

   // add chars
    cout <<  add('.', '?') << endl;

    // add strings
    string s1 ="hello";
    string s2 = "there";
    cout <<  add(s1, s2) << endl;
```

Here is the output: The template function adds all the different data types together.

```
12.40
11
m
hellothere
```

Template classes

A template class lets you specify what data type your class will hold for its instance variables. A template class only requires a header h file and no source cpp file. The template class code is to be implemented <u>after</u> the class definition in the .h header file.

A template class is defined as follows:

```
template <typename T>
class class_name
{
    private:
        variables of datatype T

    public:
        constructors
        getters
        setters
        methods
        toString

}; // don’t forget the semi-colon
```

The **template <typename T>** specifies that the T is to be substituted with a data type of your choice like char, int or double when the program runs.

There is also **template <class T>** where T is to represent a class data type rather than a primitive data type like int or double.

Both can be used there is only a slight difference between them. Most people use **template <class T>** mainly because they are substituting class data types.

We can convert our previous Calculator code into a template class. A template class calculator using our previous calculator code to do arithmetic operations add, sub, multiply and divide on 2 numbers would look like this:

```
template <typename T>
class TCalculator
{
   public:
       T add(T a, T b);
       T sub(T a, T b);
       T mul(T a, T b);
       T  div(T a, T b);

}; // don’t for get the semi colon
```

Where **T** represents the data type you specify when you instantiate the object from the template class definition.

Each template class method is implemented after the template class definition. Each method implementation starts with

```
template <typename T>
```

The method resolution operator class name must an also include the template date type <T> so that the method knows what a T is.

```
 class_name<T>::method_name()
{
}
```

Here is the template method implementation for the add method of the template calculator class

```
template <typename T>
T TCalculator<T>::add()
{
 return a + b;
}
```

To do

Type out **TCalculator** class and put into a TCalculator.h file. Below the template class definition, implement the template methods add, sub, multiply and divide.

In your Lesson12.cpp file or in a separate cpp test file called TestTCalculator.cpp. In the main method instantiate a TCalculator object and try out all the arithmetic operations. Use chars, ints, doubles and strings. Your main may look like this:

```
#include <iostream>
#include "TCalculator.h"
```

```
#include <string>

using namespace std;

int main()
{

   TCalculator<int> tc;
   cout << tc.add(3,4) << endl;   // 7

    TCalculator<string> tc2;
    cout << tc2.add("cat","dog") << endl;   // catdog

   return 0;
}
```

Note:

Some people put the template implementation in a separate cpp file. Many people try to compile the template cpp file but all they do is end up with many errors and hours of frustration. This is a bad idea, but If you do this you need to include the cpp file at the end of the template h file like this

```
#include "TCalculator.cpp"
```

Although this works but you need to remember you cannot compile the template cpp file, because the template h file calls it for you. Make sure you do not say #include "TCalculator.cpp" in your TCalculator.cpp file**.** It is best to write your template cpp file at the bottom of the template h file. You can also write all your template implementation code in-line with the template definition file like this.

```
template <typename T>
class TCalculator
{
    public:
        T add(T a, T b)
        {
        return a + b;
        }

        T sub(T a, T b)
        {
        return a - b;
        }

        T mul(T a, T b)
        {
        return a * b;
        }

        T  div(T a, T b)
        {
        return a / b;
        }
}; // don't forget the semi colon
```

Homework 12 Question 3

Instantiate a TCalculator template class for **doubles**. In a loop asking the user to type in 2 numbers and what operation they want: + - * / and display the results. You can use 'x' to exit loop.

Homework 12 Question 4

Add a accumulator instance variable to the TCalculator that will sum up all the results of the calculator. Have methods to retrieve and clear the calculator accumulator. In your main methods do some calculations print out the accumulator.

After all calculations have been made clear the accumulator and the print out its value. You need to make a Constructor that initializes the accumulator to 0;

Put your code in a file called Homework12.cpp

Simple template class

We now present a simple template class called **TData** which just stores a data value, and does operations on the data value like adding, subtraction and dividing. The operands are encapsulated (enclosed) into the class by the private variable data having specified template data type T.
The **add** method adds the encapsulated (enclosed) data value with an input parameter. Here is the **TData** template class definition.

```
/*
 * TData.h
 * template class
 */

#ifndef TData_H
#define TData_H

template <typename T>
class TData
{
private:
   T data;

public:

   TData();

   TData(T data);

   // getters
   T getData();

   // setters
   void setData(T data);

};
```

Todo

Type out the template class in a C++ header file called TData.h. A TData.cpp file is not required since the adder implementation code is to be places at the bottom of the header file like this:

```
template <typename T>
TData<T>::TData()
{
}

template <typename T>
TData<T>::TData(T  data)
{
   this->data = data;
}

// getters
template <typename T>
T  TData<T>::getData()
{
   return this->data;
}

// setters
template <typename T>
void  TData<T>::setA(TData data)
{
   this->value = data;
}

#endif // TData_H
```

You can still make a TData.cpp file but you must add

#include “Tdata.cpp”
at the bottom of the TData.h file. This is not a good thing to do to have separate template hpp and cpp fles.

todo

Type in and complete the rest of the TData implementation code. At the bottom of your Lesson9 main function or in a separate TestTData.cpp file main function, instantiate a TData class with different data types and print out the results.

```
TData<double>  td(3);
cout << td.get() << endl
td.set(5);
cout << td.get() << endl;
```

The output is as follows:

```
3
5
```

Homework12 Question 5

Make an array of TData objects.

```
TData<int> tdatas[] ={TData<int>(1),TData<int>(2),TData<int>(3),TData<int>(4),TData<int>(5)};
```

Then make a template function that will receive an array of template data objects and print them out in a loop

Here is the prototype for the printDatas template function:

```
template <typename T>
void printDatas(TData<T> datas[],int n);
```

where n is the number of elements in the array.

Homework 12 Question 6

Make a template array class called TArray that can take any data type that has methods for assigning and retrieving items. The constructor should clear the array to zero.

In a main method fill the array with numbers from 1 to 10 and print out then out.

Homework 12 Question 7

Repeat Question4 but instead use the TData template class as the data type,

Call your TData array class TDArray.

Template Interface's

We can also make an interface that is a template. We now make our **ICalculator** interface into a template ICalculator interface called **TICalculator**.

```
template <class T>
class TICalculator
{
   public:
   virtual T add(T a, T b)=0;
   virtual T sub(T a, T b)=0;
   virtual T mul(T a, T b)=0;
   virtual T div(T a, T b)=0;
};
```

We can now have the TCalculator implement the **TICalculator** interface.

```
#include "TICalculator.h"

template <typename T>
class TCalculator: public TICalculator<T>
{
```

```
public:

   T add(T a, T b);
   T sub(T a, T b);
   T mul(T a, T b);
   T div(T a, T b);

};
```

When we specify the **TICalculator** interface as the base class we must specify the data type T as well to avoid error.

class TCalculator: public TICalculator<T>

other than that the code and operation are the same.

To do

Put the TICalculator template interface in a file called TICalculator.cpp**.**

have the **TCalculator** class implement the **TICalculator** interface and run your test code. The only drawback you may have is that the string class does not have methods subtract, multiply and divide, it only has the + operation to add two strings together. You may have to avoid the string class for these operations.

Lesson 12 Home work

Question 8

Instantiate a TCalculator using the TICalculator interface class for any data type you like. In a loop asking the user to type in 2 numbers and what operation they want: + - * / and display the results. You can use ‘x’ to exit loop. Put your code in a file called Homework11.cpp

Lesson 12 Home work

Question 9

Make a TCalculator using operator methods like +.=*and / so that you can use arithmetic operations like x = a + b rather than method names like add, sub, mul and div.

Define a operator method like:

```
    T operator+(T x);
```

Use in a main program like this:

```
    int x  = a + b;
```

LESSON 13 RECURSION

When a function calls itself it is known as **recursion**. Recursion is analogues to a while loop. Most while loop statements can be converted to recursion, most recursion can also be converted back to a while loop.

The simplest recursion is a function calling itself printing out a message.

```
void printMessage()
{
    cout << "I like programming" << endl;
    printMessage();
}
```

```
I like programming
I like programming
I like programming
I like programming
I like programming
...
```

Unfortunately this program will run forever, so you will need to stop the program somehow while it is running. (Don't worry, it will stop running when it runs out of program space). We can add a counter **n** to the printMessage method so it can terminate at some point. A recursive method has a base case to stop recursion and a recursive case to continue recursion.

```
void printMessage(int n)
{
   // base case stops recursion
   if(n <=  0)
   {
       return ;
   }

  // recursive case, continues recursion
   else
   {
   cout << "I like programming" << endl;
   print_message(n-1);
   }
}
```

Now the program will print the message n times

```
I like programming
I like programming
I like programming
I like programming
I like programming
```

Every time the print_message function is called n decrements by 1.
When n is 0 the recursion stops. You may place the statement:

```
cout << "I like programming" << endl;
```

before or after the recursive call. If you put it before than the message is printed first before each recursive call.

If you put the print statement after the recursive call than the message is printed after all the recursive calls are made. There is quite a difference in program execution.
The recursive operation is very similar to the following while loop:

```
n = 5
while(n > 0)
{
        cout << “I like programming” << endl;
        n--;
}
```

You should now run the recursion function

You would call the function like this:

printMessage(5);
It will print I like programming 5 times like this:.

```
I like programming
I like programming
I like programming
I like programming
I like programming
```

n starts at 5 and counts down to 0. Before each recursive call the value of n is stored in the sequence 5,4,3,2,1. When n becomes 0 then the recursion stops. After the recursion stops the print_message function rewinds. All the previous stored values of n are restored in reverse order sequence 1,2,3,4,5 ending back where it started at the value of 5.

Todo:

Print the value of n at the top of the recursive function and at the end of the recursive function.

```
void printMessage(int n)
{
   cout << n << endl;
   // base case stops recursion
   if(n <=  0)
   {
       return ;
   }

  // recursive case, continues recursion
   else
   {
   cout << "I like programming" << endl;
   print_message(n-1);
  cout << n << endl;
   }
}
```

You will get something like this:

```
5
I like programming
4
I like programming
3
I like programming
2
I like programming
1
I like programming
0
1
2
3
4
5
```

Put the print statement after the recursive call and see what happens.

Recursion is quite powerful, a few lines of code can do so much.

Our next example will count all numbers between 1 and n. It is actually counting all the recursive loops. This example may be more difficult to understand, since recursion seems to work like magic, and operation runs in invisible to the programmer.

```
// return count
int  countn(int n)
{
  // base case
   if(n == 0)
   {
   return 0;
   }
   // recursive case
   else
    {
    return countn(n-1) + 1;
    }
}
```

count(5) would return 5 because 1 + 1 + 1 + 1 + 1 = 5

Count adds 1 to the previous returned value. n is use to count how many times we want to loop recursively.

You can run it in a program like this:

```
cout << count(5) << endl;  // 5
```

When (n == 0) this is known as the base case. When n == 0 the recursion stops and 0 is return to the last recursive call. Otherwise the **countn** function is called and n is decremented by 1.

It works like this:

```
    main calls countn(5) with  n = 5
    countn(5) calls countn(4) with n=4
    countn(4) calls countn(3) with  n=3
    countn(3) calls countn(2) with  n = 2
    countn(2)  calls countn(1) with  n = 1
    countn(1) calls countn(0) with  n = 0

   countn(0) returns 0  to count(1) since n == 0
   countn(1)  adds 1 to the return value 0  and then returns 1  to count(2)
   countn(2)  adds 1 to the return value 1  and then returns 2 to count(3)
   countn(3)  adds 1 to the return value 2  and then returns 3 to count(4)
   countn(4)  adds 1 to the return value 3  and then returns 4 to count(5)
   countn(5)  adds 1 to the return value 4  and then returns 5  to main()

   main()  receives 5 from count(5)  and prints out 5
```

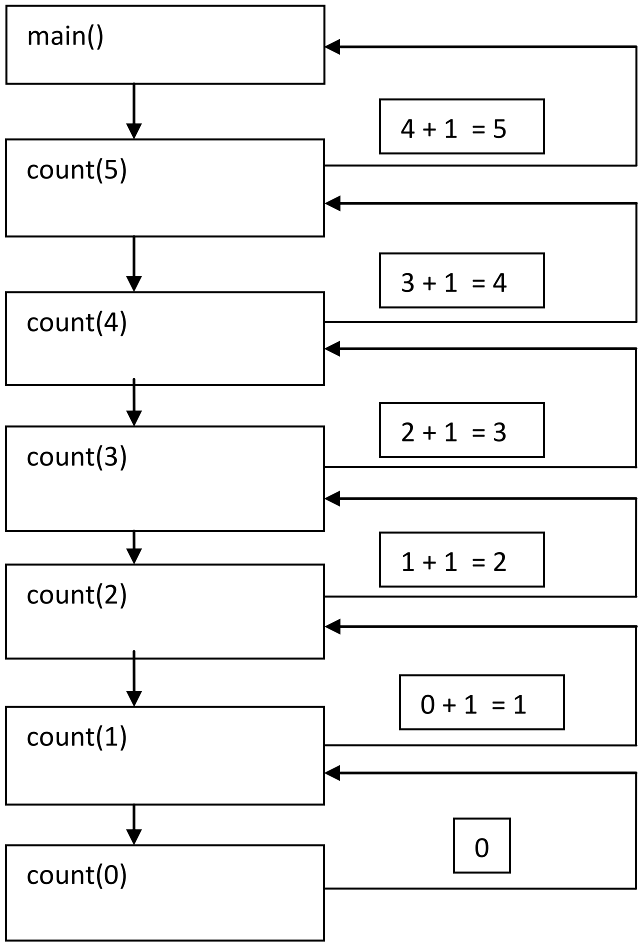
The statement **return countn(n-1) + 1** is used to call the function recursively and also acts as a place holder for the value returned by the called function.

We could rewrite the recursive part as follows:

```
int x = countn(n-1);
return  x + 1;
```

x will now receive the return value from the function call and 1 will be added to the return value and this new value will be returned to the caller.

If you can understand the above then you understand recursion. If you cannot then maybe the following diagram will help you understand.



You probably don't need to understand how recursion works right away. Sometime you just need to accept things for now then understand later. One day it will hit you when you are thinking about something else.

Basically recursion works like this:
For every recursive function call the parameter and local variables are stored. Technically they are stored in temporary memory called a stack.
Every time the recursive function returns the previous numbers that were stored are restored and now become the current number, to be used to do a calculation. The numbers are restored in **reverse** order. You must remember ever time a function is called it returns and program execution continues on the next line,

Function call/ return	**N**
call count(n-1)	5
call count(n-1)	4
call count(n-1)	3
call count(n-1)	2
call count(n-1)	1
count(n-1) returns 0	0
count(n-1) returns 0 + 1	1
count(n-1) returns 1 + 1	2
count(n-1) returns 2 + 1	3
count(n-1) returns 3 + 1	4
count(n-1) returns 4 + 1	5

The thing to remember about recursion is it always return's back where it is called, This is known as rewinding. Here are some more recursive function examples:

Sum numbers 1 to n

```
int sumn(int n)
 {
    if(n ==0)
    {
       return 0;
    }
    else
    {
      return sumn(n-1) + n;
    }
 }
```

sumn(5) would return 15

You can run it in a program like this:

```
   cout << sumn(5) << endl; // 15
```

It works similar to countn instead of adding 1 its adds the n's. sumn adds all together the previous stored n's to the recursive function returned values as it rewinds.

Before each recursive call the value of n is saved, When recursion stops when the base case is reached the recursive function returns value. The first value returned is the 0 from the base case, the 0 is added to the stored value of n which is 1 so 0 + 1 = 1 then this value is returned. Then the returned value 1 is then added to the next stored value of n which is 2 so now 1 + 2 = 3. The process continues until there is no more stored values of n available

Return value	Stored n value	return value + n
0	1	1
1	2	3
3	3	6
6	4	10
10	5	15

0+1+2+3+4+5 = 15

Our counter n serves 2 purposes a recursive counter and a number to add.

Multiply numbers 1 to n (factorial n)

We can also make a **multn** function which multiples n rather than adding n. This is basically factorial **n**. muln multiplies all together the previous stored numbers as it rewinds.

```
int multn(int n)
 {
    if(n ==0)
    {
       return 1;
    }

    else
    {
      return multn(n-1) * n;
    }
 }
```

multn(5) would return 120

 since 1*2*3*4*5 = 120

Our base case returns 1 rather than 0 or else our result would be 0;

You can run it in a program like this:

```
cout << multn(5) << endl; // 120
```

Power x^n

Another example is to calculate the power of a number x^n
In this case we need a base parameter b and an exponent parameter n.
pown multiplies all the previous stored numbers by the base b as it rewinds.

```
 int pown(int b, int n)
{
   if(n ==0)
   {
      return 1;
   }

   else
   {
     return pown(b,n-1) * b;
   }
}
```

pown(2,3) would return 8 because 2*2*2= 8 since $2^3=8$

You can run it in a program like this:

cout << pown(2,3) << endl; // 8

Every time a recursive call is made the program stores the local variables in a call stack. Every time recursive call finishes executing, the save local variables disappear and the previous local variables are available. These are the ones present before the recursive function was called. These save variables may now be used in the present calculations.

For the above example $2^3=8$ the call stack would look like this.

```
                n=0
                b=2     1
                n=1     n=1
                b=2     b=2     2
        n=4     n=2     n=2     n=2
        b=2     b=2     b=2     b=2     4
n=5     n=5     n=3     n=3     n=3     n=3
b=2     b=2     b=2     b=2     b=2     b=2     8
```

Every time the recursive function finished executing it returns a value. Each returning value is multiplied by the base b. In the above case the returning values are 1,2,4 and 8

The return value is the value from the previous function multiplied by b (2)

```
return pown(b,n-1) * b;
```

the function first returns 1 then 1 * b = 1* 2 = 2 then 2 * 2 = 4 and finally 4 * 2 = 8

efficient power x^n

A more efficient version of pown can be made relying on the fact then even n can return b * b rather than just return * b for odd n

```
int pown2(int b,int n)
 {
  if (n == 0)
  {
     return 1;
  }

  if (n %2  == 0)
  {
  return pown2(b, n-2) * b * b;
  }

  else
  {
    return pown2(b, n-1) * b;
  }
 }
```

Operation is now much more efficient 1 * 2 * 4 = 8

You can run it in a program like this:

cout << pown2(2,3) << endl; // 8

Summing a sequence

Adding up all the numbers in a sequence: **n * (n + 2) / 2**

```
   n      n *(n + 2)/2
----------------------
   0        0 + 1 (returned value)
   1        1
   2        4
   3        7
   4        12
   5        17
            -----
   Total:   42
```

```
int seqn(int n)
 {
    if(n == 0)
    {
       return 1;
    }

    else{

      return (n * (n + 2))/ 2 + seqn(n-1);
    }
 }
```

seqn(5) would return 42 because 0 + 1 + 4 + 7 + 12 + 17 = 42

You can run it in a program like this:

```
cout << seqn(5) << endl;  // 42
```

Note: 42 is the number of the universe and **n * (n + 2))/ 2 + seqn(n-1)**
Is the equation of the universe.

You can print out the sequence by modifying the **seqn** function like this:

```
int seqn2(int n)
{
   if(n == 0)
   {
      return 0;
   }

   else
   {
     int x = (n * (n + 2))/ 2;
     cout << x << endl;  // print sequence
     return (n * (n + 2))/ 2 + seqn2(n-1);
   }
}
```

You can run it in a program like this:

```
cout << seqn2(5) << endl;
```

You will get an output like this:

```
17
12
7
4
1
42
```

The sequence printed backwards and the final sum is 42

Fibonacci sequence

Recursion is ideal to directly execute recurrence relations like Fibonacci sequence

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …….

In mathematical terms, the sequence fn of Fibonacci numbers is defined by the recurrence relation.

$f_n = f_{n-1} + f_{n-2}$

with seed values

$f_0 = 0$ and $f_1 = 1$.

A **recurrence relation** is an **equation** that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

```
int fib(int n)
 {
   if (n == 0)
   {
      return 0;
   }

   else if (n == 1)
   {
      return 1;
   }

   else
   {
       return fib(n-1) + fib(n-2);
   }
}
```

Notice The recursive statement is identical to the recurrence relation

fib(10) would return 55 because 21 + 34= 55

You can run fib in a program like this:

```
   cout << fib (10) << endl;  // 55
```

Combinations

We can also calculate combinations using recursion.

Combinations are how many ways can you pick r items from a set of n distinct elements.

Call it **nCr**   (n choose r)

5C2 (5 choose 2) would be

Pick two letters from set S = {A, B, C, D, E}

Answer:{A, B}, {B, C}, {B, D}, {B, E}{A, C}, {A, D}, {A, E}, {C, D}, {C, E}, {D, E}

There are 10 ways to choose. 2 letters from a set of 5 letters. The combination formula is

nCr= n! / r!(n-r)!

The Recurrence relation for calculated combinations is:

base cases:

    nCn= 1
    nC0= 1

recursive case:

     nCr= n-1Cr + n-1Cr-1 for n > r > 0

Our recursive function for calculating combinations is:

```
int combinations(int n, int r)
{
   if (r == 0 || n == r)
   {
      return 1;
   }
  else
  {
   return combinations(n-1, r) + combinations(n-1, r-1);
 }
}
```

combinations(5,2) would return 10
You can run combinations like this:

```
cout << combinations(5,2) << endl;  // 10
```

Print a string out backwards

With recursion printing out a string backwards is easy, it all depends where you put the print statement. If you put **before** the recursive call then the function prints out the characters in reverse. Since n goes from n-1 to 0.If you put the print statement **after** the recursive call then the characters are printed not reverse since n goes from 0 to n.

```
// reverse a string
void print_reverse(string s, int n)
{
   if(n == 0)
   {
      cout << endl;
   }
```

```
  else
  {
     cout << s[n-1];
     print_reverse(s, n-1);
  }
}
```

You would call the print_reverse function like this

```
 string s = "hello";
 print_reverse(s, s.length());
```

```
olleh
```

Check if a string is a palindrome

A palindrome is a word that is spelled the same forward as well as backwards:
Like "radar" and "racecar"

```
// return true if string is a palindrome otherwise return false
bool is_palindrome( string s,  int i,  int j)
{

  if (i >= j)
     return true;
  else
  {
     if (s[i] != s[j])
        return false;
     else
        return is_palindrome(s,i+1, j-1);
  }
}
```

You would call the **is_palindrome** function like this:

```
  string s2 = "radar";
  cout << s2 << endl << (is_palindrome(s2, 0,s2.length()-1)) << endl;
```

```
Radar
1
```

```
string s3 = "apple";
cout << s3 << endl << (is_palindrome(s3, 0,s3.length()-2)) << endl;
```

```
apple
0
```

Permutations

Permutations are how many ways you can rearrange a group of numbers or letters. For example for the string “ABC” the letters can be rearranges as follows:

ABC
ACB
BAC
BCA
CBA
CAB

Basically we are swapping character and then print them out
We start with ABC if we swap B and C we end up with ACB

```
// print permutations of string s
void print_permutations(string  s, int i, int j)
{
   int k;
   char c;
   // print out permutation
   if (i == j)
   {
      cout << s << endl;
   }

   else
      {
      for (k = i; k <= j; k++) {

         // swap i and k
         c = s[i];
         s[i] = s[k];
         s[k] = c;

         // recursive call
         print_permutations(s, i + 1, j);

         // put back, swap i and k
         c = s[i];
         s[i] = s[k];
         s[k] = c;
      }
   }
}
```

You would call the **print_permutations** function like this:

```
String s4 = "ABC";
print_permutations(s4, 0,s4.length()-1);
```

```
ABC
ACB
BAC
BCA
CBA
CAB
```

Combination sets

We have looked at combinations previously where we wrote a function to calculate home many ways you can choose r letters from a set of n letters.

nCr   n choose r

Combinations allow you to pick r letters from set S = {A, B, C, D, E}

n = 5 r = 2 nCr   5C 2

Answer:{A, B}, {B, C}, {B, D}, {B, E}{A, C}, {A, D}, {A, E}, {C, D}, {C, E}, {D, E}

We are basically filing a seconded character array with all possible letters up to r.

Start with ABCDE we would choose AB then AC then AD then AE etc.
We use a loop to traverse the letters starting at n =0, and fill the comb string.
When n = r we then print out the letters stored in the comb string.

```
void print_combinations(string s, char combs[],
     int start, int end, int n, int r)
{
   int i = 0;
   int j = 0;

   // current combination is ready to be  printed
  if (n == r)
   {
      for (j = 0; j < r; j++)
         cout <<combs[j];
      cout << endl;
      return;
   }

   // replace n with all possible elements.
   for (i = start; i <= end && end - i + 1 >= r - n; i++)
   {
      combs[n] = s[i];
      print_combinations(s, combs, i+1, end, n+1, r);
   }
}
```

You would call the print_combinations function like this:

string s5 = "ABCDE";
char combs[5+1] = {0};
print_combinations(s5, combs,0,s5.length()-1,0,2);

```
A B
A C
A D
A E
B C
B D
B E
C D
```

The difference between combinations and permutations is that in a combination you can have different lengths in the set where as in permutations they are the same length in the set.

Determinant of a matrix using recursion.

In linear algebra, the determinant is a useful value that can be computed from the elements of a square matrix. The determinant of a matrix A is denoted det(A), detA , or |A

In the case of a 2 × 2 matrix, the formula for the determinant is:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

For a 3 × 3 matrix A, and we want the s formula for its determinant |A| is

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & I \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

$$= aei + bgf - ceg - bdi - afh$$

Each determinant of a 2 × 2 matrix in this equation is called a "minor" of the matrix A. The same sort of procedure can be used to find the determinant of a 4 × 4 matrix, the determinant of a 5 × 5 matrix, and so forth.

Our code actually follows the above formula, calculating and summing the miners.

```
 // calculate determinant of a matrix
float determinent(float matrix[3][3], int size)
{
   int c;
   float det=0;
   int sign=1;
   float b[3][3];
   int i,j;
    int m,n;

   // base case
   if(size == 1)
   {
      return (matrix[0][0]);
   }
   else
   {
      det=0;
      for(c=0; c<size; c++)
      {
         m=0;
         n=0;
         for(i=0; i<size; i++)
         {
            for(j=0; j<size; j++)
            {
               b[i][j] = 0;
               if(i!=0 && j!=c)
               {
                  b[m][n] = matrix[i][j];
                  if(n<(size-2))
                  {
                     n++;
                  }
```

```
                else
                    {
                        n=0;
                        m++;
                    }
                }
            }
        }
        det = det + sign*(matrix[0][c]*determinent(b,size-1));
        sign = -1*sign; // toggle sign
    }
  }
  return (det);
}
```

You call and run the determinant function like this:

```
  float m[3][3] = {{6,1,1},{4,-2,5},{2,8,7}};

      cout << "det = " << determinent(m,3) << endl;
```

```
-306
```

There are many more recursive examples, too numerous to present.
If you do all the following to do questions you will be a recursive expert.

LESSON 13 HOMEWORK

Question 1
Print out an array using recursion.
int printArray(int a[],int n);

Question 2
Add up all numbers in an array using recursion.
int add(int a[],int n);

Question 3
Print out even numbers in an array using recursion.
void printEven(int a[],int n);

Question 4
Print out odd numbers in an array using recursion.
void printOdd(int a[],int n);

Question 5
Print out an array backwards using recursion.
void printBackwards(int a[], int n);

Question 6
return largest number in an array using recursion
int largest(int a[],int n);

Question 7
return smallest number in an array using recursion
int smallest(int a[],int n);

Question 8
Write a recursive function called **void reverse_string(char s[], int n)** that reverses a char string in place. The recursive string receives the char string and outputs the string in reverse. No printing is allowed.

Question 9
Write a recursive function called **void reverse_string(string s, int n)** that reverses a sting in place. The recursive string receives the string and outputs the string in reverse. No printing is allowed. You need to use **substr** since you cannot replace individual letters in a string.

Question 10
Write a recursive function called **bool is_palindrome2(string s)** that receives just a string parameter and returns true if the string is a palindrome. You will need to use **substr** to reduce the string as you test the first and last letters.

Question 11
Write a recursive function **int search_number(int a[], Int n, int x)** that searched for a number in an array and returns the index of the number if found otherwise returns -1 if not found.

Question 12
Write a recursive function **bool search_digit(int d, int x)** that searches for a digit inside a number and return true of the number if found otherwise returns false if not found.

Question 13
Write a recursive function called **int sum_digits (int d)** that adds up all the digits in a number of any lengths. The recursive function receives an int numbers and returns the sum of all the digits.

Question 14
Write a recursive function called **void format_number(char* s, int n)** that can insert commas in a number. For example 1234567890 becomes 1,234,567,890

Question 15
Write a recursive function **bool is_even(int n)** that return true if a number has even count of digits or false if the number of digits is odd.

Question 15
Write a recursive function **bool is_odd(int n)** that return true if a number has odd count of digits or false if the number of digits is odd.

Question 16
Write a recursive function **void print_binary(int d)** that would print a decimal number as a binary number. A binary number just has digits 0 to 1.
Where a decimal number has digits 0 to 9. The decimal number 5 would be 0101 in binary, since 1*1 + 0* 2 + 1* 4 + 0 *8 is 10. We are going right to left.

To convert a decimal number to binary You just need to take mod 2 of a digit and then divide the number by 2

```
5%2 = 1  🡐 1
5/2 = 2
2 %2 = 0   🡐 0
2/2 = 1
1 %2 = 1   🡐 1
1/2  = 0
0 %2 = 0   🡐 0
```

We are done so going backwards
5 in binary is 0 1 0 1

Question 17
Write a recursive function **bool is_prime(int x, int n)** that returns true if a number is prime otherwise false. Where x is the number and n is the recursive counter the x – 1.
A prime number can only is divides evenly by itself. 2,3,5,7, are prime numbers. You can use the mod operator % to test if a number can be divided evenly by itself. 4 %2 = 0 4 can be divided evenly by 2 so there for 4 is not a prime number.

Put all your functions in a cpp file called Lesson13.c Include a main function that tests all the recursive functions.

PROJECTS

Project 1 IntArray Class

Make an IntArray class to store a int values in an internal array called items. Make a default constructor that makes an empty array. Make another constructor that takes in the initial size of the array. Make another constructor the receives an ordinary array. You need to copy the elements in the receiving array to your internal array. Make another constructor that receives your IntArray. Again, you need to copy the elements in the receiving IntArray to your internal array. Make

methods to access array elements by array index. Make operational methods to add items to the end, insert at a certain index, and remove at a certain index. When adding and inserting items the internal array should just increase in size by 1. When removing items from the internal array just shift the other vales down and set the last value to 0. Make operational method to sort the array ascending and search for values. Use bubble sort to sort the array and use binary search to search for items in the internal array when it is sorted. You can find the code for bubble sort and binary search on the internet. Lastly make a toString method to print out the array elements enclosed in square bracket's like this: [9 4 9 3 6 4 8]
Make a TestArray class with a main method to test all the methods of your Array class or alternately for convenience put the main method inside your IntArry class.

Project 2 Int Matrix class

Make a Matrix class that has rows and column variables and an two-dimensional array of the specified rows and columns. Make a default constructor to make an empty Matrix of rows and columns. Have private variable to store rows and columns. Make another constructor the receives an ordinary two-dimensional aray. You need to copy the elements in the receiving array. Make another constructor that receives your IntMatrix. Again, you need to copy the elements in the receiving IntMatrix. Make setters and getters to access the matrix elements. make a to string method that will print oy the matrices Make operational methods to add, subtract, multiply, divide, transpose and rotate matrices by a specified rotation. Use operator functions for convenience.
Make a TestMatrix class with a main method to test all the methods of your Matrix class

Project 3 Spelling Corrector

Read in a text file with spelling mistakes, find the incorrect spelled words and offer corrections. The user should be able to choose the correct choice from a menu. Look for missing or extra letters or adjacent letters on the keyboard. Download a word dictionary from the internet as to check for correct spelled words. Use a **vector** or **set** to store the words. Store the correct spelled file.

Project 4 MathBee

Make a Math bee for intermixed addition, subtraction, multiplication and division single digit questions. Use random numbers 1 to 9 and use division numbers that will divide even results. Have 10 questions and store results in a file. Keep track of the users score. You studied how to make random numbers in the C mini lessons. In C++ you need to use the following include statements:

```
#include<cstdlib>
#include<ctime>

// seed random number generator
srand((unsigned int)time(0))

// get random number 1 to 10
int x = (rand() % 10) + 1;
```

Project 5 Quiz App

Make a quiz app with intermixed multiple choice, true and false questions. You should have a abstract Question super class and two derived classes MultiipleChoice and TrueAndFalse. Each derived class must use the abstract methods to do the correct operation. Store all questions in one file. Store the results in another file indicating the quiz results.

Project 6 Phone Book App

Make a phone book app that uses a **map** to store phone numbers, emails and names. You need an **Contact** class to store name, phone number and email. Make a toString() method or use friend functions for printing out contact info. Make copy constructor , assignment and equal operators. You should be able to add, view, search and remove contacts and exit as menu operations. Contacts need to be displayed in alphabetically orders by name. Offer to lookup contacts by name or by phone number. Contacts should be stored in a file, read when app runs, and saved with app finish running. Bonus: add a menu option to view contacts sorted by phone numbers or by emails.

Project 7 Appointment App

Make an Appointment book app that uses a **map** to store Appointments. You need an Appointment class to store name, description and time. You should be able to view, add, delete, and scroll up and down appointments as menu operations. Appoints need to be displayed in chronological orders. Appointments should be stored in a file, read when app runs, and saved with app finished running.

Project 8 template Array Class

Make the IntArray class to be a template Array class TArray so that it can store any data type.
Make a TestTArray class with a main method to test all the methods of your TArray class.

Project 9 template Matrix class

Make the IntMatrix class to be a template Matrix class TMatrix so that it can store any data type. Use operator functions for convenience.
Make a TestTMatrix class with a main method to test all the methods of your TMatrix class.

Project 10 template Arithmetic Classes

Make Generic Add, Sub, Mult and Divide template classes so each has a overloaded method to do arithmetic operations on any data type. Have an template abstract super class called Operations to represent all the Operation classes. Incorporate the classes in your template Calculator of Lesson 6. Rename you template calculator to TCalculator2. Use operator functions for convenience.

Project 11   Grocery Store App

Make a Grocery Store App where Customers can purchase items. Preferred customers get a discount. After all items have been entered a receipt is printed.

Step 1: Item class

Make a Item class with private variables product name, quantity ordered, price and discountPrice.
If the item is not a discount item then the discount price is 0.
Make a constructor that will receive the item name, quantity, price and discount price.
Make getters and setters for each instance variable

Make a formatted toString method that will return item name price quantity discount price surrounded by round brackets and extension price like this:

```
Carrots        2     1.29  (.89)    2.58  (1.78)
```

Step 2: GroceryStore class

Make a GroceryStore class that will store items bought, total items bought, that total's the order and print out a receipt.

The Grocery Store class will store the customer's name, and all items bought in an vector <Item> called items.
The Grocery store constructor will receive the customer name and create the vector <Item> of items.
The grocery store will have a method to add an item object called add.
The grocery store class will also print out a receipt using a method called printReceipt.

The Grocery store class will have a getTotal method to return the total of all items. The getTotal method can also be used to print out the receipt total.

All instance variables are private and you cannot have any getters and setters.

Step 3: DiscountStore class

The DiscountStore class inherits from the GroceryStore class.
The DiscountStore receives the CustomerName and sends the CustomerName to the GroceryStore super class.

If the customer is a preferred customer then the DiscountStore class is used.
The DiscountStore class will calculate discount percent, and count of discount items and total of all items using the discount price rather than the regular price.
The discount class will override the getTotal class of the grocery store class.
The discount store class will also print a receipt showing the number of discount, items and the discount percent obtained.

Step 4   GroceryApp class.

The GroceryApp class is the main class where the cashier enters the customer items, bought.
The cashier will ask if the customer is a preferred customer. if it is a preferred customer then the DiscountStore class is used else the Grocery Store class
Is used.

The cashier will enter the items bought. Once all items have been enters then the receipt is printed out.

You will need to store a list of products in a file to simulate the entering of products or make an array of items like this:

```
Item items[] = {Item(“apples”,2,1.26,1.08)
Item(“oranges”,2,1.26,1.08),
Item(“carrots”,2,1.26,1.08)
Item(“apple”,2,1.26,1.08)};
```

The file format will be like this

```
Customer name
Preferred or not preferred
Number of products
Item name, quantity, price, discount price
```

Example file:

```
Tom Smith
Preferred
3
```

Carrots , 2.49, 1.78, 2
Fish,12.67, 11.89,3
 Milk 4.89.3.75, 2
In either case the items will be added to the store.

The main method will have a menu as follows:

(1) add items to Grocery store
(2) add items to Discount store
(3) print receipt
(4) exit program

END