

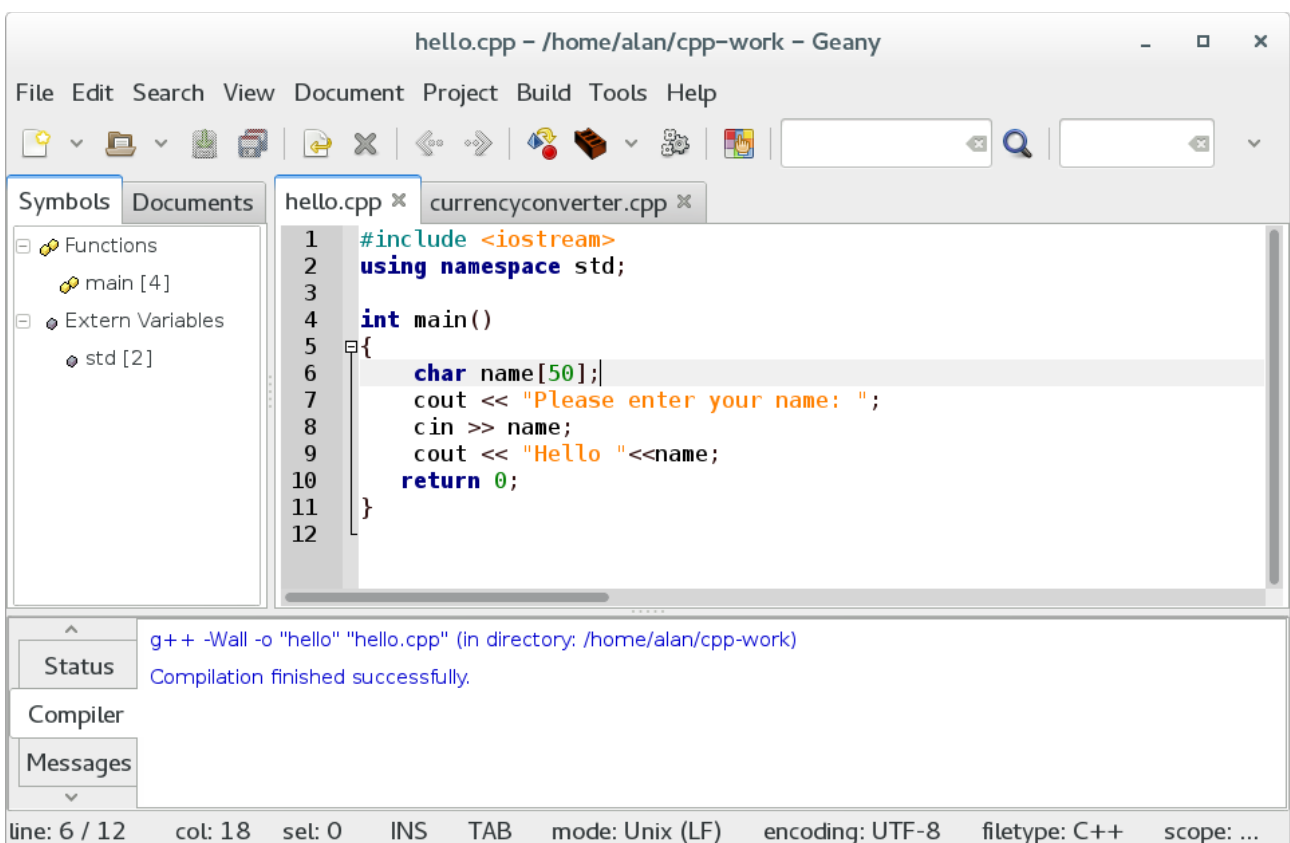
C++ Programming with Linux

Dr Alan Crispin

C++ is a programming language that runs on a variety of platforms, such as Windows, Mac OS, and Linux. This primer will demonstrate how to use C++ to develop application code using the Debian Jessie Linux operating system. One of the best ways to teach a new programming language is by using example programs. This primer consists of a collection of sample programs designed to introduce a beginner to the C++ programming language.

Integrated Development Environment (IDE)

An Integrated Development Environment (IDE) is a single program for software development typically consisting of a source code editor and tools to help compile files and build executables. Geany is a lightweight simple IDE with a clean interface for developing C++ programs. A screen shot of the Geany IDE is shown below.



The basic work flow is:-

1. Create a new file (File ->New)
2. Write your C++ code
3. Save your code with the .cpp extension (File->Save or use the save file icon)
4. Build your code (Build->Build)
5. Execute your code using the run icon or press F5 or select Build->Execute

In the above screen shot, the code is saved as `hello.cpp` and when this is built, a compiler message appears in a box at the bottom of the window. To execute the code press F5 or press the run current file icon. A separate terminal window is invoked.

Compiler

A compiler is a computer program that transforms source code written in a high level language such as C++ into instructions (machine object code) understood by a computer's central processing unit (CPU). Linux uses the open source C++ compiler called `g++`.

Notice in the compiler window of Geany you see the command:-

```
g++ -Wall -o "hello" "hello.cpp"
```

Here `g++` is the C++ compiler and `-Wall` is the (warn all) flag that enables most warning messages, which is extremely important for debugging your code. The `-o` option tells the compiler to create an executable file called `hello` and `"hello.cpp"` is the actual source code file. Understanding and fixing warnings and errors produced using the `-Wall` flag can help with debugging code which does not compile.

With Geany there are **compile** and **build** menu options. Compiling is the process of converting a C++ code file into a machine level object file (`.o` extension). Building is the process of converting C++ code to an executable and this involve compiling and linking. Linking is the process of combining object code with libraries.

Example Program 1: Structure

The sample program below shows the basic syntax of a C++ program.

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

int main()
{
    char name[50];
    cout << "Please enter your name: ";
    cin >> name;
    cout << "Hello "<<name;
    return 0;
}
```

Output:

```
Please enter your name: alan
Hello alan
```

```
-----
(program exited with code: 0)
Press return to continue
```

The program asks the user to enter their name and then prints the word “hello” followed by the name that was entered. To do this it uses the standard console output stream called *cout* (in-conjunction with the insertion operator `<<`) to print information to the screen and the standard input stream called *cin* (in-conjunction with the extraction operator `>>`) to read information from the keyboard into a character array called `name` (see below for more information on arrays). To use *cout* and *cin* we must add the `#include <iostream>` header to the code. We also need to add the line “`using namespace`

std;” or we would have to use `std::cin` for the standard input from the console and `std::cout` for the standard output to the console. Here, `::` is the scope resolution operator (more on this later). The line `int main()` is where program execution begins. This is called the main function which should return an integer value. The `return 0;` line terminates the `main()` function and causes it to return the value 0 to the calling process. Notice that curly brackets are used to define a block of code. Every opening curly bracket “{” must have a closing curly bracket “}”. In C++ programming lines are terminated by a semicolon.

Variables and Data Types

A variable is a name for a piece of memory that can be used to store information. Each variable must have a data type. So, for example, to define an integer variable called “i” we would use the code line:-

```
int i =4;
```

Here `int` (short for integer) defines the type, “i” is the variable name and it stores 4. The main data types used by C++ are `bool` (Boolean value), `char` (single character value), `float` (floating point value), `double` (double precision floating point number), `void` (valueless value) and `wchar_t` (wide character). Some of the basic types can be modified using type modifiers: signed, unsigned, short and long.

C++ allows enumerated (enum) types to be defined. An example is:-

```
enum PaintColour {blue, green, orange};  
PaintColour aColour(orange);
```

An enumerated type allows names to be associated with a restricted range of integer values. In this example, the names blue, green and orange are associated with the integer values 0, 1 and 2 respectively.

Control Flow Statements

C++ provides control statements for altering the flow in which statements are executed. These include conditional branching and looping. Conditional branching structures can be used for decision making and include:-

if statement

```
if(boolean_expression)  
{  
    // code statements will execute if the boolean expression is true  
}
```

if...else statement

```
if(boolean_expression)  
{  
    // code statements will execute if the boolean expression is true  
}  
else  
{  
    // code statements will execute if the boolean expression is false  
}
```

switch case statement

```
switch(expression){  
    case constant-expression:  
        //code statements;  
        break;  
    case constant-expression:  
        //code statements;  
        break;  
  
    default: //Optional  
        //code statements;  
}
```

Notice that the switch statement allows a variable to be tested for equality against a list of values. It uses four keywords in the C++ language, namely; switch, case, break and default. Remember to use break statements between each of the case statements otherwise the flow of execution falls through to the next case statement.

Lines in source code which start with // are single line comments and are ignored by the compiler. You can also use block (multi-line) comments with /**/.

You use loop structures when you need to execute a block of code a number of times.

while loop

```
while(condition)  
{  
    //code statements;  
}
```

The while loop repeats a statement or group of statements while a given condition is true.

for loop

```
for ( initial condition; end condition; increment )  
{  
    //code statements;  
}
```

The for-loop executes a sequence of statements multiple times using initial and end conditions together with an increment value.

do...while loop

```
do  
{  
    //code statements;  
} while(condition);
```

The do while loop is like a while statement except that it tests the condition at the end of the loop body so that the block of code in the do statement is executed at least once. C++ also has a “goto” control statement which allows an unconditional jump to a labelled statement in the same function. Using “goto” statements is not recommended and should be avoided.

We now have enough knowledge to start developing some simple C++ applications.

Example Program 2: Currency Converter

This example shows how to write a simple currency converter for four currencies. The user enters the number of pounds to convert and is presented with four options, namely convert to US dollars, Euros, Russian rubles or Australian dollars. This example demonstrates how to use the switch statement.

```
#include <iostream>
using namespace std;
int main()
{
    double data;
    int choice;
    float pounds;
    cout << "Currency converter";
    cout << "Please enter the number of pounds that you want to convert.\n";
    cout << "£";
    cin >> pounds;
    cout << "Convert to... \n";
    cout << " 1 - US Dollars \n";
    cout << " 2 - Euros \n";
    cout << " 3 - Russian Rubles \n";
    cout << " 4 - Australian Dollars \n";
    cout << ": ";
    cin >> choice;
    switch(choice)
    {
        case 1:
            data = (pounds * 1.5);
            cout << "Converted from GB pounds to US Dollars.\n";
            cout << "The result was " << data << ". \n";
            break;
        case 2:
            data = (pounds * 1.39);
            cout << "Converted from GB pounds to Euros.\n";
            cout << "The result was " << data << ". \n";
            break;
        case 3:
            data = (pounds * 80.0);
            cout << "Converted from GB pounds to Russian Rubbles.\n";
            cout << "The result was " << data << ". \n";
            break;
        case 4:
            data = (pounds * 1.94);
            cout << "Converted from GB pounds to Australian Dollars.\n";
            cout << "The result was " << data << ". \n";
            break;
        default:
            data = 0;
            cout << "Invalid.";
            break;
    }
    return 0;
}
```

Functions

A function is a group of statements designed to perform a specific operation packaged together with a name so that can be called by a program. The general format of a C++ function definition is:

```

return_type function_name(parameter1, parameter2, .. )
{
    //body of the function
}

```

The return_type is the data type returned by the function. The function_name is the identifier by which the function can be called. The parameters (e.g. parameter 1, parameter2) allow arguments to be passed to the function with each parameter separated by a comma. Each parameter must have a type followed by an identifier (e.g. int h, int w). The block of statements between the curly brackets { and } specify what the function actually does.

The C++ standard library provides numerous built-in functions that your program can call. For further information see:- <http://en.cppreference.com/w/cpp/header>

Example Program 3: Function (Prime Number Checker)

A prime number is an integer number which is greater than 1 and which is divisible only by 1 or itself. The example code below shows how to write a function to check if an integer number is a prime number or not.

```

#include<iostream>
using namespace std;

bool is_prime(int number) {
    int i,count=0;
    if(number==1 || number==2) return true;

    if(number%2==0) return false; //if divides by 2 then not a prime

    for(i=1;i<=number;i++)
    {
        if(number%i==0) count++; //if prime then count should increment by 2
    }

    if(count==2)return true;
    else return false;
}

int main()
{
    int n;
    cout<<"Enter number to check if prime ";
    cin>>n;
    bool result = is_prime(n);
    if(result==true)
    {
        cout<<" Number " <<n <<" is a prime" <<endl;
    }
    else
    {
        cout<<" Number " <<n <<" is not a prime"<<endl;
    }
    return 0;
}

```

Output:

```

Enter number to check if prime 5
Number 5 is a prime

```

In this example, the function is called “is_prime” and takes an integer number as a parameter and returns a Boolean (true or false) result. The code inside the function checks to see if the integer number passed to the function is a prime number or not. It does this by performing a set of checks using the remainder operator “%”. Let's take an example of passing the number 5 to the “is_prime” function. The first check tests if the number is 1 or 2 as both these number are primes. If the number entered was either 1 or 2 the function would terminate at this point and return a true Boolean value. The second check “if(number%2==0)” tests if the number has a remainder of zero when it is divided by 2. In our case, $5\%2=1$ and so we go onto the next check. This is a for-loop which counts up the number of times that a zero remainder occurs when the number is divided by all numbers up to its own value. If the number is a prime this count should be two as a prime can only be divided by 1 and itself (i.e. a count of 2). Consequently, if the count is 2 we return true (the number is a prime) else we return false (the number is not a prime).

Recursion

Recursion is when a function calls itself. At first this may seem like a never ending loop as it would seem on the surface that a function that calls itself will never finish. However, in practice we make sure there is a check to see if a certain condition is true and in that case exit (return from) the function.

Consider writing a function for calculating the factorial of an integer number, say 4. The factorial is

$$4! = 4*3*2*1 = 24$$

We can see that $4! = 4 \times 3!$ and that $3! = 3 \times 2!$ and so on. We are defining the problem in terms of itself which is the basis for recursion. The next code example shows how to write a recursive factorial function.

Example Program 4: Recursion

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

int factorial(int n){
    if(n!=1){
        return(n * factorial(n-1));
    }
    else return 1;
}

int main(){
    cout << "The factorial of 4 is " << factorial(4) << endl;
    return 0;
}
```

Output:

The factorial of 4 is 24

```
-----
(program exited with code: 0)
Press return to continue
```

Pointers

A pointer is a memory location or address of a variable say, x. It allows you to pass around the location where a variable is stored and gives you the ability to change the original value of the variable, in this case x.

Example Program 5: Pointers

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

int main()
{
    int x=3;
    int *mypointer; //create a pointer using the dereferring operator
    mypointer = &x; //store the memory location of the x variable in the pointer

    cout << "x = " << x << endl;
    cout << "mypointer address = " << &mypointer << endl; //address
    cout << "value stored by mypointer =" << *mypointer << endl; //value

    *mypointer = 10;
    cout << "value stored by mypointer =" << *mypointer << endl; //value

    return 0;
}
```

The output from the pointer example is:-

```
x = 3
mypointer address = 0x7fff003b84c8
value stored by mypointer =3
value stored by mypointer =10
-----
(program exited with code: 0)
Press return to continue
```

You can pass around “mypointer” just like it was a variable as demonstrated by the code lines

```
*mypointer = 10;
cout << "value stored by mypointer =" << *mypointer << endl; //value
```

Pointers are used because they are very memory efficient as you do not need to create copies of a variable. You can pass a pointer to a function when you want to change the original value.

Pass by reference (or pointer)

Passing a pointer to a function is known as “pass by reference”. This is very useful as it allows a function to modify a local variable. For example, consider the following example program which shows that the PassByRef() function is able to modify the local variable (x) defined inside main().

Example Program 6: Pass by Reference (Pointer)

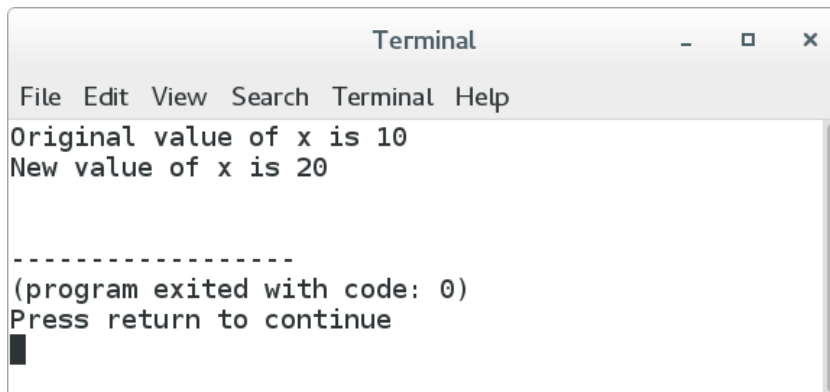
```
#include <iostream>
using namespace std;

void PassByRef(int &x) { //pass by reference
    x = 20;
}
```



```
int main()
{
    int x = 10;
    cout<<"Original value of x is "<<x <<endl;
    PassByRef(x);
    cout<<"New value of x is "<<x <<endl;
    return 0;
}
```

Output:



```
Terminal
File Edit View Search Terminal Help
Original value of x is 10
New value of x is 20

-----
(program exited with code: 0)
Press return to continue
```

Remember that “pass by reference” means that you are passing a reference to the actual variable and so when you are changing it, say in a function, you are changing the actual value of the variable at the pointer location.

Pass by Value

Pass by value means pass a copy of the variable to the function so that the original value of the variable does not get changed. See example below.

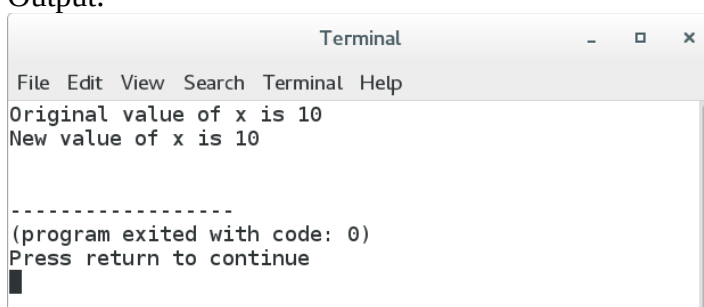
Example Program 7: Pass by Value (Copy)

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

void PassByValue(int x) { //pass by value
    x = 20;
}

int main()
{
    int x = 10;
    cout<<"Original value of x is "<<x <<endl;
    PassByValue(x);
    cout<<"New value of x is "<<x <<endl;
    return 0;
}
```

Output:



```
Terminal
File Edit View Search Terminal Help
Original value of x is 10
New value of x is 10

-----
(program exited with code: 0)
Press return to continue
```

The original value of x (x=10) is not changed by the PassByValue(x) function as only a copy of x is passed to the body of the function.

Arrays

An array is a collection of variables of the same type. When declaring an array you have to specify the data type of the elements in the array and the number of elements required. For example,

```
float arr1[10];
```

Here the array called “arr1” can store ten floating point values. You can initialize array elements as follows:-

```
float arr1[10] = { };           // all elements are 0
```

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0	1	2	3	4	5	6	7	8	9

The first element of an array always has an index of 0. You can also specify the values of elements as shown below.

```
float arr1[10] = {1.1, 2.2, 1.4}; //unspecified elements are set to zero
```

You can also initialise an array using values:

```
int array[] = { 1,2,3,4,5}; // integer array
```

```
char array2 ="code"; //char array
```

Remember that array data structures set aside a contiguous area of memory for one specific type of variable.

Example Program 8: Array Example

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

int main() {
    const int MAX_STUDENTS=5;
    float studentGrades[ MAX_STUDENTS ] = { 0.0 };
    for (int i=0; i<MAX_STUDENTS; i++) {
        cout << i << " " << studentGrades[i] << endl;
    }
    return 0;
}
```

The console output is:-

```
0 0
1 0
2 0
3 0
4 0
```

Example Program 9: Calculate Average (passing an array to a function)

Build, run and test the source code below which calculates the average for a set of numbers entered at the keyboard. The code example uses a function called average which has a float return type (the calculated average) and accepts an array parameter. The array is passed by value in this example as we do not want to change the array values, just calculate the average.

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

//function to calculate average
float average(float theArray[], int size){
    float result = 0.0;
    for(int i = 0; i < size; i++){
        result += theArray[i];
    }
    return result/size;
}

int main(){

    float myarray[10];
    int length =0;
    cout << "Enter 10 integers:" << endl;

    while( length != 10 )
        { cin >> myarray[length++]; }

    float av = average(myarray,10);
    cout << endl << "average: " << av << endl;

    return 0;
}
```

Multidimensional Arrays

Multidimensional arrays can be thought of as "arrays of arrays". For example, you would declare a two dimensional integer array called "arr2d" as:-

```
int arr2d = [4][4];
```

This is a 4-by-4 two-dimensional table of integer elements with each element accessed using a row, column index as shown below.

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

Makefiles

Makefiles are generally used to build applications consisting of multiple files and when linking with libraries. You can think of a makefile as a script for organising the compilation of a project. Consider a project which consists of a set of C++ source code files with each one responsible for an arithmetic integer operation i.e. add.cpp, subtract.cpp and multiply.cpp. We would define the function prototypes

using a functions.h header file as follows.

```
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
```

Assume that a main.cpp source file uses these operations as shown below.

```
#include <iostream>
#include "functions.h"
using namespace std;

int main(){
    cout << "Adding 1+2 = " << add(1,2) << endl;
    cout << "Subtract 3-1 = " << subtract(3,1) << endl;
    cout << "Multiply 3*2 = " << multiply(3,2) << endl;
    return 0;
}
```

The console commands to compile and run the application would be:-

```
g++ -Wall main.cpp add.cpp subtract.cpp multiply.cpp -o calc
./calc
```

We can write the following makefile to compile the application.

```
calc: main.o add.o subtract.o multiply.o
    g++ main.o add.o subtract.o multiply.o -o calc
main.o: main.cpp
    g++ -c main.cpp
add.o: add.cpp
    g++ -c add.cpp
subtract.o: subtract.cpp
    g++ -c subtract.cpp
multiply.o: multiply.cpp
    g++ -c multiply.cpp
clean:
    rm *.o calc
```

The console commands to compile and run the application using the makefile would be

```
make
./calc
```

Makefile Structure

So as you can see from above, a makefile is simply a script with a set of rules to describe how to produce a target output from the various files that make up the project. The make command reads the makefile and, using the compilation rules within it, produces an executable. The structure of a simple makefile consists of three parts, namely;

- the link rule
- the compilation rule
- the clean-up rule.

Each of these rules has the same basic format:

```
file: component-file1 component-file2 component-file3 ...
```

Link rule

The link rule starts the makefile. In the above example the link rule is:

```
calc:  main.o add.o subtract.o multiply.o
      g++ main.o add.o subtract.o multiply.o -o calc
```

Here “calc” is the name of the executable. The files main.o, add.o, subtract.o, and multiply.o are compiled object files created by the compiler but are not executable. The first line tells the compiler that “calc” will need to be recompiled if any of these machine object files change. On the second line, g++ is the name of the compiler and the files main.o, add.o, subtract.o, and multiply.o are used to produce the output executable file called “calc” (the -o <file> option places the output into <file>).

Compile Rule

In the above example, the compile rule is:

```
main.o: main.cpp
      g++ -c main.cpp
add.o:  add.cpp
      g++ -c add.cpp
subtract.o: subtract.cpp
      g++ -c subtract.cpp
multiply.o: multiply.cpp
      g++ -c multiply.cpp
```

It tells the compiler how to create the individual compiled object (.o) files from the various .cpp source files. The -c option tells the compiler to produce a compiled object (.o) file (i.e. suppress any linking just compile stated sources file)

Clean Rule

The clean rule is optional and in the above example it is:

```
clean:
      rm *.o calc
```

It allows you to use the terminal command “make clean”. The clean command removes all compiled object files (*.o) and the executable file (calc) in the working directory. You can use the recursive force -rf option (with caution) which would recursively remove compiled object files from sub-directories if they existed.

The Geany IDE compiles single files and so you need to use the Build → Make menu item command when compiling an application with multiple files. It assumes that a file called "makefile" is stored in the same folder as your application files and uses this to compile the application. The next example shows how to do this.

Example Program 10: Makefiles

Create a new directory (e.g. makefile-example) and then write the source files called main.cpp, add.cpp, subtract.cpp and multiply.cpp together with a functions header file called functions.h as shown below.

```

//main.cpp
#include <iostream>
#include "functions.h"
using namespace std;

int main(){

    cout << "Makefile example" << endl;
    cout << "Adding 1+2 = " << add(1,2) << endl;
    cout << "Subtract 3-1 = " << subtract(3,1) << endl;
    cout << "Multiply 3*2 = " << multiply(3,2) << endl;
    return 0;
}

//functions.h
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);

//add.cpp
#include "functions.h"
int add (int a, int b){
    return a+b;
}

//subtract.cpp
#include "functions.h"
int subtract(int a,int b){
    return a-b;
}

//multiply.cpp
#include "functions.h"
int multiply(int a,int b){
    return a*b;
}

```

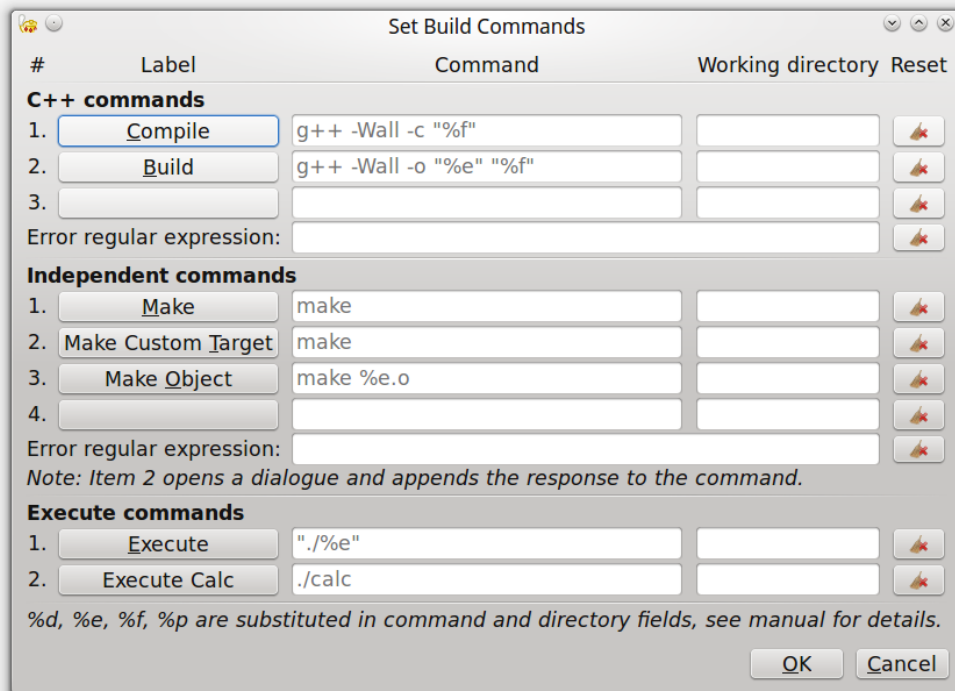
To compile these files with Geany write the following makefile and store it in the same directory.

```

calc:  main.o add.o subtract.o multiply.o
      g++ main.o add.o subtract.o multiply.o -o calc
main.o: main.cpp
      g++ -c main.cpp
add.o: add.cpp
      g++ -c add.cpp
subtract.o: subtract.cpp
      g++ -c subtract.cpp
multiply.o: multiply.cpp
      g++ -c multiply.cpp
clean:
      rm *.o calc

```

To build the project with Geany, open the main.cpp file then choose Make in the Build menu (shift+F9). Then create an execute command using Build → Set Build Commands with the name “Execute Calc” and set the terminal command to ./calc (see screen shot below). Once this is done you run the application using the menu item Build-> Execute Calc



The compilation output should be:

```
make (in directory:)
g++ -c main.cpp
g++ -c add.cpp
g++ -c subtract.cpp
g++ -c multiply.cpp
g++ main.o add.o subtract.o multiply.o -o calc
Compilation finished successfully.
```

The console output should be:

```
Makefile example
Adding 1+2 = 3
Subtract 3-1 = 2
Multiply 3*2 = 6

-----
(program exited with code: 0)
Press return to continue
```

The makefile compiles each of the source code files (.cpp extension) to a compiled object file (.o extension). The arithmetic operator files (add.cpp, subtract.cpp and multiply.cpp) are said to be dependencies for the target main.cpp file.

Why use MakeFiles?

Makefiles can take advantage of the fact that when you have many source files (.cpp files) in a project, it is not necessary to recompile all the files when you make a change to only one of these. You often only need to recompile one or a small subset of the files. That is, if a large project has a number of dependencies, you may only need to compile one of a number of files rather than everything which saves time.

Tutorial on makefiles	https://www.cs.bu.edu/teaching/cpp/writing-makefiles/
Makefile manual	http://www.gnu.org/software/make/manual/

Header Files

You will have noticed in the above example that it used a header file (.h extension). Typically when you write more advanced C++ applications you write your code as a set of classes (see below) which consist of two files; a header file (.h file extension) and a source code file (.cpp file extension). The source code (.cpp file) typically contains the implementations of all methods in a class. The corresponding header (.h file) contains variable declarations and function (method) prototypes. The purpose of the header (.h) files is to export "services" to other source code .cpp files.

Data Structures

A data structure is a group of data members combined together under a single name. We use the "struct" keyword to define a structure type. You can access data member using the dot (.) operator as shown in the following example which reads the names and birth years of three people and then displays these.

Example Program 11: Data Structures (struct)

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

struct person {
    string name;
    int year;
};

int main ()
{
    person p[3]; //array to store three people
    string n_str;

    for (int i = 0; i < 3; i++)
    {
        cout <<"Please enter your name ";
        getline(cin,p[i].name);
        cout <<"Please enter your birth year ";
        getline(cin,n_str);
        stringstream(n_str)>>p[i].year;    //convert string to int
    }
    cout<<"The birth years for the following people are: "<<endl;

    for (int i = 0; i < 3; i++)
    {
        cout<<p[i].name <<" was born "<<p[i].year<<endl;
    }

    return 0;
}
```

Output:


```
Please enter your name Jeff
Please enter your birth year 1996
Please enter your name John
Please enter your birth year 1997
Please enter your name Jane
Please enter your birth year 1995
```

The birth years for the following people are:

```
Jeff was born 1996
John was born 1997
Jane was born 1995
```

```
-----
(program exited with code: 0)
Press return to continue
```

The data structure called “person” has two data members called *name* and *year*. The *name* data member is of type string. The standard C++ library provides a string class and to use it you have to add the `#include <string>` header to the code. The *year* data member is of type int. The code defines an array to store three objects of structure type person. The first for-loop allows a user to enter three names and birth years. The second for-loop displays the entered information in the console. Notice that in the first loop a function called `getline()` is used which extracts characters from the *cin* stream and stores them into a string (`p[i].name` in this case). The `getline()` is used again to extract the numbers entered for the birth year and stored them in the string called *n_str* (number string). The `stringstream()` class is used to convert the string *n_str* into an integer value.

Classes and Objects

A class is a collection of related data members and functions under a single name. It contains data members, and functions (often called methods) and represents a template to model some entity. An object is an instance of a class. Classes are defined using the keyword “class” and actually define a type. This is best explained using an example.

Consider that we wish to model a person (the entity). We would create a class called person and then create people instances (objects) from it. The person class is the template from which we create people. To write the person class we need to think about what attributes a person has and what methods we may need. All people have a name and age and so we will use these. Notice that the “Person” class is difference to the person data structure defined above because it has methods as well as data members.

Example Program 12: Person Class

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

using namespace std;

class Person {
private:
    int age;
    string name;
public:
    Person(); //constructor
    ~Person(); //destructor
    void set_name(string the_name); //sets the name
    string get_name(); //returns the name
```

```

        int get_age(); //returns the age
        void set_age (int new_age); // sets the age
};

Person::Person()
{
    cout <<"Object created on the heap "<<endl;
    name ="default";
    age = 0;
}

void Person::set_name(string the_name)
{
    name=the_name;
}

string Person::get_name()
{
    return name;
}

void Person::set_age(int the_age){
    age=the_age;
}

int Person::get_age(){
    return age;
}

Person::~~Person()
{
    cout<<"Object destroyed" <<endl;
}

int main () {

    //Create object on the heap
    Person *student1 = new Person();
    student1->set_name("Jodie");
    student1->set_age(22);
    cout << "Student1 name = " << student1->get_name()<<endl;
    cout << "Student1 age = " << student1->get_age()<<endl;
    delete student1;
    return 0;
}

```

Output:

```

Object created on the heap
Student1 name = Jodie
Student1 age = 22
Object destroyed
-----

```

```

(program exited with code: 0)
Press return to continue

```

Memory in a computer is divided into what is known as the stack and heap. The stack is automatically managed memory while the heap memory is that part of RAM used for dynamic memory allocations. When an object is created on the stack it is automatically deleted when it goes out of scope. The way you create objects and then access member functions on the stack and heap is different as shown below.

Stack

```

Person student1;
student1.set_age(24);

```

When an object (e.g. `student1`) is created on the stack you use the dot operator (`.`) to access member functions.

Heap

```
Person *student1 = new Person();  
student1 → set_age(24);
```

When an object is created on the heap you use a pointer (e.g. `*student1`) and the *new* operator. The arrow operator (`→`) is used to access member functions. It is recommended that objects are created on the heap but when you do this it is your responsibility as a programmer to delete the object when you have finished with it. To do this you use the *delete* operator.

Notice that the class variables *name* and *age* have been declared as *private*. This means that they are private to the class and cannot be accessed except through the getter (e.g. `get_age`) and setter (e.g. `set_age`) methods. This is known as data encapsulation or data hiding. Another operator, the scope resolution operator “`::`”, has been introduced which is used to access the members declared in the class. So, `Person::set_age` allows us to access the set age method previously declared in the class. The class declaration could have been put into a separate header file but in this example I have kept things together (simple) for demonstration purposes. Also notice that the class name, in this case `Person`, is treated as a type in the main function.

I have also introduced a constructor. A constructor is a special method which has the same name as the class name (in the case `Person`) which has no return type and is used for initialisation. In this example, I am using the constructor to set default values for the person name and age. If you comment out the following lines in the main function:

```
student1.set_name("Jodie");  
student1.set_age(22);
```

you will see these default values in the console out. The `~Person()` method is another special method called a destructor. A destructor has the class name preceded with a tilde (`~`). The purpose of the destructor is to free up resources used by the object (e.g. clean up memory allocated to arrays or objects inside the class using the `delete` operator).

By introducing classes, we have touched on what is quite a complex subject, that of object oriented programming (OOP). However, we needed to do this as we often want to use external libraries (see multimedia section below) in our code and these will almost certainly be made up of a series of classes. One of the major concepts of OOP is inheritance. Inheritance involves creating a new class from a base class. The derived class inherits the public member functions (methods) of the base class and can then add its own member functions on top of these.

File Input Output

C++ provides a standard library for handling file streams which is called `<fstream>` which provides the ability to read and write files. A file stream object must be opened when reading and writing data. For writing data you use *ofstream* (output file stream) which is used like the *cout* stream. For reading data you use an *ifstream* (input file stream) which is used like the *cin* stream.

Example Program 13 Writing Data to a File (Saving)

```
// writing to a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    ofstream myfile ("example.txt");
    if (myfile.is_open())
    {
        myfile << "The cat sat on the mat\n";
        myfile << "The dog began to bark\n";
        myfile.close();
    }
    else cout << "Unable to open file";
    return 0;
}
```

Output: A file called example.txt is created in the current working directory

Example Program 14 Reading Data from a File

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

int main () {
    string line;
    ifstream myfile ("example.txt");
    if (myfile.is_open())
    {
        while ( getline (myfile,line) )
        {
            cout << line << '\n';
        }
        myfile.close();
    }

    else cout << "Unable to open file";

    return 0;
}
```

Output:

The cat sat on the mat

The dog began to bark

(program exited with code: 0)

Press return to continue

Multimedia

SFML is a cross-platform (Windows, Linux and Mac) library which can be used for the development of C++ multimedia and game applications. It was created by the French programmer Laurent Gomila in C++. Its license makes it free even for commercial use. See web sites www.sfml-dev.org and <https://github.com/LaurentGomila>

To install the SFML library in Debian Jessie you use the commands:-

```
su -
aptitude install libsFML-dev
```

Example Program 15: Draw Graphics with the SFML Library

Using Geany write the source file below and save it as "main.cpp". This is one of the sample sfml tutorial programs found at <http://www.sFML-dev.org/tutorials/2.0/start-linux.php>

```
#include <SFML/Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Green);

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        window.clear();
        window.draw(shape);
        window.display();
    }
    return 0;
}
```

To compile this file at the terminal you would use:

```
g++ -c main.cpp
```

Then you must link the compiled machine file (main.o) to the SFML libraries in order to get the final executable. Each of the five SFML modules (system, window, graphics, network and audio) have their own library. To link an SFML library, you must add "-lsfml-xxx" to the compiler command, for example "-lsfml-graphics" for the graphics module. To compile and run the project at the command line you would use:-

```
g++ main.o -o sfml-app -lsfml-graphics -lsfml-window -lsfml-system
./sfml-app
```

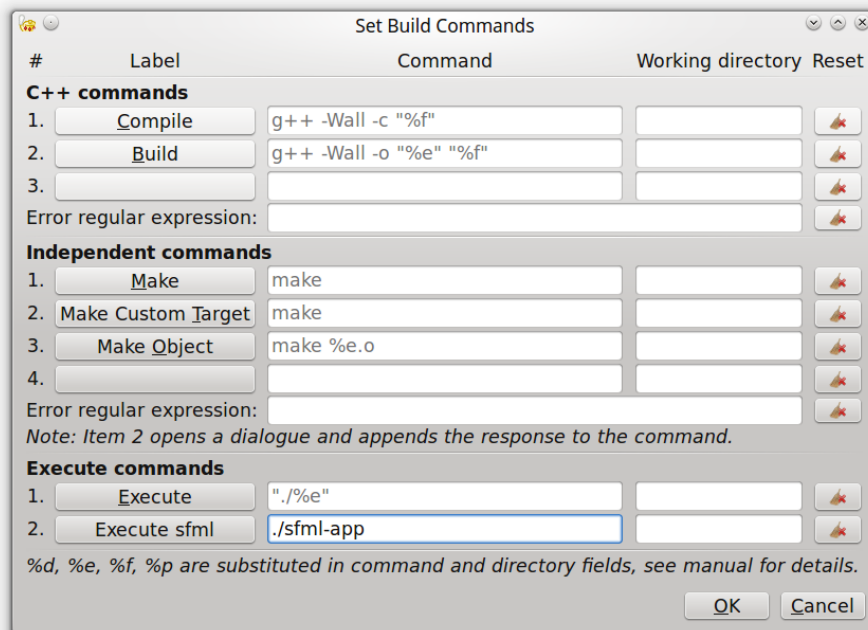
To compile and execute the main.cpp file using Geany you need to create the makefile as shown below.

```
sfml-app: main.o
    g++ main.o -lsfml-graphics -lsfml-window -lsfml-system -o sfml-app
main.o:
    g++ -c main.cpp
clean:
    rm *.o sfml-app
```

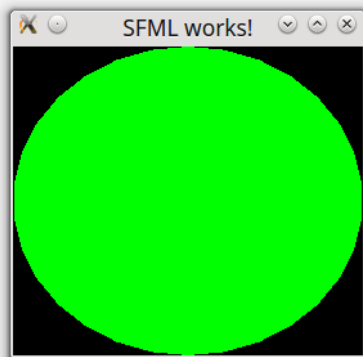
Save this as "makefile" in the current working directory where main.cpp is stored. The next step is to make the project using Build → Make (or shift+F9). You should see the message "compilation

successfully finished” in the Geany's status window. Finally you need to create a special purpose execute command to run the application. To do this go to Build → Set Build Commands

Then create an “Execute sfml” entry using the terminal command `./sfml-app` as shown below.



To run the sfml application use Build → Execute sfml. You should get the output shown below.



Playing Audio Files

SFML provides a class called `sf::Sound` for playing small sounds with audio data loaded from `sf::SoundBuffer` (memory). The example program below plays the “tone.wav” file which should be located in the current working directory.

Example Program 16: Play Sound

```
#include <SFML/Audio.hpp>
#include <iomanip>
#include <iostream>
#include <string>

void playSound(std::string the_sound)
```

```

{
    // Load a sound buffer from a wav file

    sf::SoundBuffer buffer;
    if (!buffer.loadFromFile(the_sound))
        return;

    // Create a sound instance and play it
    sf::Sound sound(buffer);
    sound.play();

    // Loop while the sound is playing
    while (sound.getStatus() == sf::Sound::Playing)
    {
        // Leave some CPU time for other processes
        sf::sleep(sf::milliseconds(100));

        // Display the playing position
        std::cout << "\rPlaying... " << std::fixed << std::setprecision(2) <<
sound.getPlayingOffset().asSeconds() << " sec  ";
        std::cout << std::flush;
    }
    std::cout << std::endl << std::endl;
}

int main()
{
    // Play a sound
    std::string the_sound;
    the_sound = "tone.wav";
    playSound(the_sound);

    // Wait until the user presses 'enter' key
    std::cout << "Press enter to exit..." << std::endl;
    std::cin.ignore(10000, '\n');

    return EXIT_SUCCESS;
}

```

Build the project using the makefile below.

```

sfml-app: main.o
    g++ main.o -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio -o
sfml-app
main.o:
    g++ -c main.cpp
clean:
    rm *.o sfml-app

```

Then run the code using the “Execute sfml” menu item which executes the terminal command

```
./sfml-app
```

You should hear a continuous audio tone. SFML provides another class for playing music called `sf::Music`. It does not load all the audio data into memory but instead streams it dynamically from a source file and so for memory efficiency reasons should be used for playing compressed audio files. For more information see <http://www.sfml-dev.org/tutorials/2.0/audio-sounds.php>

A tutorial on how to play an audio tone generated from a sine wave can be found at:-

<https://github.com/LaurentGomila/SFML/wiki/Tutorial:-Play-Sine-Wave>

Further information

There are many good C++ tutorial sites on the web. Some of these are listed below.

<http://www.tutorialspoint.com/cplusplus/>

<http://www.cplusplus.com/>

<http://www.learncpp.com/>

C++ Keywords

<http://www.learncpp.com/cpp-tutorial/14c-keywords-and-naming-identifiers/>

C++ Standard Library

http://www.tutorialspoint.com/cplusplus/cpp_standard_library.htm

SFML Tutorials

<http://www.sfml-dev.org/tutorials/2.1/>

Alan Crispin (Updated 15-05-15)