Computational Mechanics

Kyle Perez

December 15, 2020

Preface

My Preface

-Kyle Perez

Contents

1 The Logic of Programming

Before we can run, we must first learn to walk. We cannot simply start our adventures in programming by writing the most complex programs that our mind can imagine. We start small, with the basics, and then building up and up.

In vein of this, let's start by looking over the most basic aspects of programming.

1.1 Compiling Code

Before we can run any code, beyond using our built-in knowledge, we need a compiler. In essence, a compiler can translate code in one language into code in another language. The particular workings of this are complicated. For the most part, code is translated either directly or indirectly into Assembly, and then into binary.

Given that binary code proves very hard to read due to its files being incredibly long and only composed of a small handful of characters, we should avoid programming in binary if at all possible. Similarly, Assembly provides a similar limitation; it is complicated and difficult to read. However, we have at our disposal a wide variety of high and low level computer languages that are far easier to read.

So, before we can do anything, we must install a compiler for our given languages. The GNU Compiler Collection (gcc) is a freely available, open source set of compilers that handle C, C++, and Fortran. Similarly, Python is a freely available programming language with its own compiler packed on in. Mathematica is a proprietary software that is includes a built-in way to compile the language.

I leave it to you to install these yourself, but most Linux distributions have a way to install gcc directly via the terminal, while Python has a .deb or related file available. Mathematica can be downloaded and setup as a bulk program.

A compiler will not catch errors in logic, but it will fail to produce an executable file if there is a serious error in your program. A good compiler will give you an idea of what is wrong, but sometimes the messages can be misleading; the best way to go about debugging the inevitable errors is by looking at the first error produced.

In order to compile a program you must first write it. There are many ways to write the necessary files, but the program files can be written into a simple text editor and saved with a particular ending. For example, a Python script must be saved with a .py at the end, a C++ program a .cpp, and a Fortran program a .f90 at the end[1]. gedit is a freely availible text editor that does a reasonable job on Linux, but I leave the alternatives up to you to research. We will go over more specifics in later sections.

Regardless, for this section, I will use C++ for my various code examples, but see later sections for code in different languages. Suffice it to say, much of the idea behind everything is the same.

1.2 Variables

A variable is an element that we can assign a value to. We are free to specify a name to a variable, and assign it an appropriate value.

[1]These are not the only file endings that need to be used.

Ax an example, saving this file as *my_program.cpp*:

```
#include <iostream>

using namespace std;

int main() {
    int a = 1;

    cout << "The value of a is " << a << endl;

    return 0;
}//end main
```

Compiling and running this produces the output:

```
g++ -o my_program my_program.cpp
./my_program
The value of a is 1
```

There are restrictions on what a variable can be named, but these restrictions are often down to character length limits and first character limits. Some languages are case insensitive. However, it's important to note that it is almost always better to be very specific with your variable names, so that you can tell the purpose of a variable just by its name.

As an example, if we were to code a RPG (video game), a reasonable name for the variable that contains your player's name would be *player_name*, and not something like *i* or *sjd*.

Note how I assigned the value of *a* using the '=' equals sign. In practically all programming languages, the equal sign is the operator that assigns values to a given variable.

However, note that I had placed the characters 'int' before declaring 'a'. This is because variables in C++ require you to declare the type of data that a variable will be holding. This is not uniform across all languages; both Python and Mathematica (which are high-level languages) don't have this restriction. Regardless, these languages still have the same basic types of data. Let's go over the most common of them.

1.2.1 Integers

Integers are whole numbers with a plus or minus sign. Programming languages represent them as sets of 0s and 1s which causes some issues for large numbers that we will ignore for the time being.

From a mathematical perspective, the Integers form a ring with respect to our common addition and multiplication. This means that the product of two integers is an integer, and the sum of two integers is an integer. This means that we can do basic mathematical operations on integers. However, we cannot divide by them under most circumstances.

Importantly, integers are precise. That is, there is no uncertainty attached to it, which is an issue that we will encounter in floating point numbers. However, they can be a bit cumbersome and slow when it comes to multiplying large integers together.

Regardless, integers are a vital part of every programming language, and they can be used to index sets of numbers and to provide useful ways of storing data about how many times we do something.

1.2.2 Floating Point Numbers

Floating point numbers (or just floats) have a lot in common with integers in that they are numbers. In particular, floats more closely line up with real numbers[2] in general. We can add, subtract, multiply, divide, and perform almost any elementary operation on them.

However, floats have some amount of uncertainty related to them, given that not every fraction can be expressed in binary in a finite amount of characters. However, floats can be used to make calculations faster than if they were done with integers. In particular, floats can be used to describe function values, or anything else that real numbers have a use for.

1.2.3 Boolean Variables

Boolean variables can have one of two values: true or false. Much like integers, they are exact, however, they are clearly limited in what they can do. Regardless, they still have their uses, for example, Boolean values can be used in control structures, where they are of prime importance.

1.2.4 Strings

Strings are lines of data that consist of typeable characters. We can a string to a single variable, of build it up over time using a program we build. Strings are important to use when considering non-binary options for a function, for example if we build a program that outputs a *csv* file, we need to provide a string to the function so that it can serve as a title for the csv.

1.3 Comparing Variables

Given two variables, we are capable of comparing them in a variety of ways. This has obvious uses, for example, if we want some code to run only if one variable is less than another. We will investigate cases like this further, but for now let's stick to the logic of comparisons.

1.3.1 Order

By Order, I mean comparison using $<$ or $>$. That is, less than or greater than. When comparing variables, these expressions evaluate to Boolean types: true or false.

Let's examine an example (where the filename is obvious):

```
#include <iostream>

using namespace std;

```

[2]More accurately with rational numbers, but I digress

```
int main() {
    bool a = (2 < 3);//Evaluates to true

    //true is equal to 1, false is equal to 0
    cout << "The value of a is " << a << endl;

    a = (2 > 3);//Evaluates to 0

    cout << "The value of a is " << a << endl;

    return 0;
}//end main
```

Compiling and running this produces the output:

```
g++ -o basic_compare basic_compare.cpp
./basic_compare
The value of a is 1
The value of a is 0
```

Note that the boolean value is either a 0 or a 1, corresponding to a respective false and true. This is true in C++.

1.3.2 Equals

In a somewhat similar manner, we can compare variables of many types using the equality operator '=='[3]. We use this double equals sign since the single version is used by the assignment operator. We can use this to compare two variables of multiple types.

See the following code:

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

using namespace std;

int main() {
    bool a = (2 == 2);

    cout << "The value of a is " << a << endl;

    a = (2 == 3);

    cout << "The value of a is " << a << endl;

```

[3]Note: Fortran uses a different system which we will discuss later. .EQV. can compare logicals, while .EQ. can compare everything else.

```
    string my_string = "Hello";
    string my_second_string = "World";

    a = (my_string == my_string);
    cout << "The value of a is " << a << endl;

    a = (my_string == my_second_string);
    cout << "The value of a is " << a << endl;

    return 0;
}//end main
```

Compiling and running this produces the output:

```
g++ -o compare compare.cpp
./compare
The value of a is 1
The value of a is 0
The value of a is 1
The value of a is 0
```

Note that we can compare not just Boolean values with the equality, but also strings, where two strings are considered equal if and only if all their elements are the same in the correct order. This generalizes to comparing other types of data. However, be warned about floating point numbers; the imprecision can mess up the equality comparison, so for comparing floats we may want to take a safer approach for comparison.

1.3.3 Not Equal to

While we have '==' to compare objects, we also have '!=' that returns true when two variables are not the same. This is the inverse of the equality operator[4].

Observe:

```
#include <iostream>

using namespace std;

int main() {
    bool a = (2 != 3);

    cout << "The value of a is " << a << endl;

    a = (2 != 2);

    cout << "The value of a is " << a << endl;
```

[4] In C++ we can use the '!' to negate a bool. IE: bool a = true; a = !a implies a == false.

```
    return 0;
}//end main
```

Compiling and running this produces the output:

```
g++ -o neq neq.cpp
./neq
The value of a is 1
The value of a is 0
```

1.3.4 And

The And operator allows us to combine two Boolean statements and means we can form compound statements. For example, if we want to check if a number is between two other numbers, we need to use a compound and statement.

See:

```
#include <iostream>

using namespace std;

int main() {
    bool a = ( (1 < 3) and (3 < 4) );

    cout << "The value of a is " << a << endl;

    a = ( (1 < 3) and (1 == 2) );

    cout << "The value of a is " << a << endl;

    return 0;
}//end main
```

Compiling and running this produces the output:

```
g++ -o and and.cpp
./and
The value of a is 1
The value of a is 0
```

Note that the And operator works like the intersect operation in Set Theory.

More importantly, in order for a compound And statement to be true, every substatement must be true.

1.3.5 Or

The Or operator lets us combine two Boolean statements like the And statement. Or gives true if at least one of the two arguments is true. This means that if both arguments are true, the Or returns true[5].

An example of a use of an Or statement would be counting random numbers that lie between two disjoint intervals.

Observe:

```
#include <iostream>

using namespace std;

int main() {
    bool a = ( (1 < 3) or (3 != 3) );

    cout << "The value of a is " << a << endl;

    a = ( (3 !=3) or (1 != 1) );

    cout << "The value of a is " << a << endl;

    return 0;
}//end main
```

Compiling and running this produces the output:

```
g++ -o or or.cpp
./or
The value of a is 1
The value of a is 0
```

The Or operator works just like the union operation in Set Theory. Moreover, in order for a compound Or statement to be true, at least one substatement must be true.

1.4 Procedural Control

So far when exploring our ability to compare variables and such, we only really assigned Boolean values to variables, which clearly has limited usefulness.

The real power of this ability to compare statements is in a whole set of commands that allow us to selectively execute blocks of code, or to repeatedly execute blocks of code.

1.4.1 If and Else

If statements are many and numerous throughout codes. Moreover, when I use the word 'If', I include Else If and Else statements.

[5]The Xor operator is like the Or operator, but returns false if both arguments are true; it can be built from And, Or, and Not statements.

These are very clear and understandable ways of executing code. In particular, an If statement checks to see if its argument is true. If it is true, it executes it, if it is false, it doesn't execute it.

Then there is the Else statement. An Else must be paired with an If, with the If statement coming first. If the If statement fails to fire, then the code contained in the Else statement will run.

Finally, there is the Else If statement. From the internal perspective, an Else If statement is just an If statement nested inside an Else Statement. So, the Else If statement must be linked with an If statement. If an If statement fails, the program goes down the Else If statements to find the first that is true and triggers only the first one that is true. If there is an Else statement and no If or Else If statements are true, then the Else statement triggers.

See the following code that accepts user input and triggers code based on your input:

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

using namespace std;

int main() {
    string my_favorite_language;

    cout << "Which is your favorite programming language?";
    cout << "C++ or Java?" << endl;
    cout << "My favorite programming language is ";
    getline(cin, my_favorite_language);

    if (my_favorite_language == "C++") {
        cout << "That's a good language." << endl;
    }
    else if (my_favorite_language == "Java") {
        cout << "That's a shame." << endl;
    }
    else {
        cout << "You didn't pick one of ";
        cout << "the languages I wanted." << endl;
    }//end if

    return 0;
}//end main
```

Compiling and running this produces different output depending on what you enter, so running the program three times:

```
g++ -o my_favorite_language my_favorite_language.cpp
./my_favorite_language
Which is your favorite programming language? C++ or Java?
```

```
My favorite programming language is C++
That's a good language.

./my_favorite_language
Which is your favorite programming language? C++ or Java?
My favorite programming language is Java
That's a shame.

./my_favorite_language
Which is your favorite programming language? C++ or Java?
My favorite programming language is Python
You didn't pick one of the languages I wanted.
```

1.4.2 While Loops

Next, instead of selecting which code to run based off of some condition, let's continue to execute code based off of a condition remaining true, we call these While Loops[6].

A While loop will execute code so long as its condition is satisfied. This means that it is very easy to create an infinite loop if we aren't careful.

Let's revisit our code from the last example and make it so that if we don't put in a certain input the code will continue to run.

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

using namespace std;

int main() {
    bool my_answer_is_valid = false;
    string my_favorite_language;

    while (my_answer_is_valid == false) {
        cout << "Which is your favorite programming language?";
        cout << "C++ or Java?" << endl;
        cout << "My favorite programming language is ";
        getline(cin, my_favorite_language);

        if (my_favorite_language == "C++") {
            cout << "That's a good language." << endl;
            my_answer_is_valid = true;
        }
        else if (my_favorite_language == "Java") {
            cout << "That's a shame." << endl;
```

[6]Fortran is such an old language that we call such loops 'do while' loops. Not to be confused with 'do' loops.

```
            my_answer_is_valid = true;
        }
        else {
            cout << "You didn't pick one of ";
            cout << "the languages I wanted." << endl;
        }//end if
    }//end while

    return 0;
}//end main
```

Compiling and running this produces different output depending on what you enter, so running the program three times:

```
g++ -o while_my_fav_language while_my_fav_language.cpp
./while_my_fav_language
Which is your favorite programming language? C++ or Java?
My favorite programming language is Python
You didn't pick one of the languages I wanted.
Which is your favorite programming language? C++ or Java?
My favorite programming language is C++
That's a good language.
```

Note that we are able to nest loops within one another, so there could be a situation in which we have while loops within one another.

1.4.3 For Loops

While While loops continue so long as a a condition is true, For Loops start with an initial condition, and will iterate, updating the counting variable of the loop with every loop.

For Loops are especially powerful for building tables of data and making csv files and a significant portion of our code will use For Loops given the focus on computational physics.

See this simple program that prints the iterating variable:

```
#include <iostream>

using namespace std;

int main() {

    for (int i = 0; i < 4; i = i + 1) {
        cout << "My value for i is " << i << endl;
    }//end for

    return 0;
}//end main
```

See what we get:

```
g++ -o my_loop_counter my_loop_counter.cpp
./my_loop_counter
My value for i is 0
My value for i is 1
My value for i is 2
My value for i is 3
```

1.4.4 Break

The Break command is very powerful, and it allows us to break out of loops early for whatever reason that we may choose. As fate would have it, the Break command is relatively quick to execute, which is nice to know, which means that exiting While loops can often be done via this command more efficiently.

See the following code that we've executed previously about our favorite programming language, just with break statements instead:

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

using namespace std;

int main() {
    bool my_answer_is_valid = false;
    string my_favorite_language;

    while (my_answer_is_valid == false) {
        cout << "Which is your favorite programming language?";
        cout << "C++ or Java?" << endl;
        cout << "My favorite programming language is ";
        getline(cin, my_favorite_language);

        if (my_favorite_language == "C++") {
            cout << "That's a good language." << endl;
            break;
        }
        else if (my_favorite_language == "Java") {
            cout << "That's a shame." << endl;
            break;
        }
        else {
            cout << "You didn't pick one of ";
            cout << "the languages I wanted." << endl;
        }//end if
    }//end while
    return 0;
}//end main
```

Note the output:

```
g++ -o break_my_fav_language break_my_fav_language.cpp
./break_my_fav_language
Which is your favorite programming language? C++ or Java?
My favorite programming language is Python
You didn't pick one of the languages I wanted.
Which is your favorite programming language? C++ or Java?
My favorite programming language is C++
That's a good language.
```

Still, we can use this for more than just While Loops. If we really wanted to, we could use Break statements in any type of loop we'd like to. For example, here's a basic For Loop that does this:

```
#include <iostream>

using namespace std;

int main() {

    for (int i = 0; i < 4; i = i + 1) {
        cout << "My value for i is " << i << endl;

        if (i == 2) {
            cout << "I'm breaking the statement" << endl;
            break;
        }//end if

    }//end for

    return 0;
}//end main
```

We get

```
g++ -o break_my_loop_counter break_my_loop_counter.cpp
./break_my_loop_counter
My value for i is 0
My value for i is 1
My value for i is 2
I'm breaking the statement
```

1.4.5 Switch

Switch is an alternative to using If-Else statements for integers and characters. As it turns out, If-Else statements are relatively computationally expensive compared to the Switch statement (although not all languages support it). We can use the Switch statement along with Break statements in order to accomplish a slightly-faster program.

See the following code:

```
#include <iostream>

using namespace std;

int main() {

    int my_fav_number;

    cout << "My favorite number is ";
    cin >> my_fav_number;

    switch(my_fav_number) {
        case 1://IE: my_fav_number == 1
            cout << "That is my favorite number too!" << endl;
            break;
        case 2:
            cout << "I hate 2." << endl;
            break;
        default:
            cout << "I'm indifferent to that number." << endl;

    }//end switch

    return 0;
}//end main
```

Running the output a few times:

```
g++ -o my_fav_number my_fav_number.cpp
./my_fav_number
My favorite number is 1
That is my favorite number too!

./my_fav_number
My favorite number is 2
I hate 2.

./my_fav_number
My favorite number is 3
```

```
I'm indifferent to that number.
```

It's unfortunate that we can't compare strings using C++'s Switch statement, so we can make do with If-Else statements, but clearly we can use them to compare integers, which is of some usefulness.

1.5 Comments and Formatting

Commenting code is important. So far the programs that I have presented are simple and nearly self-explanatory, however this almost never going to be the case. Moreover, note the following: code that you write yourself will (hopefully) make sense to you. But this need not be the case for someone else who reads your code. To make your life easier, comment your code liberally and explain as much as possible[7].

Similarly, we should also strive to organize and space out our code properly so it is nice and easy to read. If we group everything up in one giant list, things will be difficult to read, and thus difficult to see what is going on and difficult to debug.

See the following example of uncommented and poorly formatted code for a numerical integrator

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <vector>
#include <utility>
#include <random>
using namespace std;
double rand_double(double lower_bound, double upper_bound) {
        return (upper_bound - lower_bound)*
            ( (double)rand() / (double)RAND_MAX) + lower_bound;
}
double integrate_1d(double function(double), double lower_bound,
    double upper_bound, int num_steps) {
        double integral_value = 0.0;
        double step_size = (upper_bound-lower_bound)
            /static_cast<double>(num_steps);
        vector<double> partition;
        vector<double> partition_samples;
        for (int i = 0; i < num_steps + 1; ++i) {
                if (i == 0) {
                        partition.push_back(lower_bound);
                }
                else {
                        partition.push_back(partition[i-1]
```

[7]For example, I have taken up the practice of commenting a notice for when a loop, if statement, or function ends, since it can be difficult in C++ to keep track of the brackets. Other languages need not have this issue, but it still doesn't hurt.

```
                        + step_size);
                }
        }
        for (int i = 0; i < partition.size() - 1; ++i) {
                partition_samples.push_back(
                    rand_double(partition[i], partition[i+1])
                );
        }
        for (int i = 0; i < partition_samples.size(); ++i) {
                integral_value = integral_value
                    + function(partition_samples[i]) * step_size;
        }
        return integral_value;
}
double my_function(double x) {
        return sin(x);
}
int main() {
        cout << "My function integrates to ";
        cout << integrate_1d(my_function, 0.0, 1.0, 1000) << endl;
}
```

And for commented and better formatted (although no doubt there is better formatting out there[8]):

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <vector>
#include <utility>
#include <random>

using namespace std;

// A function that generates a random double
// between lower_bound and upper_bound
double rand_double(double lower_bound, double upper_bound) {

        return (upper_bound - lower_bound)
            *( (double)rand() / (double)RAND_MAX) + lower_bound;

}//end rand_double

```

[8]The small size of the pages makes long lines of code go past the page; it makes code look a bit bad in LaTeX. Thankfully C++ doesn't care about line breaks and only semicolons.

```
// A function that integrates a 1D function. The output is a double.
        // It takes in a function that accepts and
            //outputs doubles as arguments.
        // lower_bound and upper_bound are the upper and lower
            //bounds of the integral respectively
        // num_steps is the size of the partition that the program
            //will create to sum over.
double integrate_1d(
    double function(double),
    double lower_bound,
    double upper_bound,
    int num_steps
) {

        // We initialize the integral to 0.
        double integral_value = 0.0;

        // step_size is the distance between two elements
            //of the partition; it is uniform
        double step_size = (upper_bound-lower_bound)
                          /static_cast<double>(num_steps);

        // partition is a vector that records all
            //partition elements in order
        vector<double> partition;
        // partition_samples is a vector that holds a double
            //that is between two elements of the partition
        vector<double> partition_samples;

        // We create a partition with num_steps + 1
            //so we do num_steps in the Riemann Sum
        for (int i = 0; i < num_steps + 1; ++i) {

                // For our first step we insert
                    //lower_bound as the 0th element
                if (i == 0) {
                        partition.push_back(lower_bound);
                }
                // Otherwise, we insert the previous partition
                    // element plus the step size
                else {
                        partition.push_back(
                            partition[i-1] + step_size
                        );
            }//end if

        }//end for

        // partition_samples[i] is a random number between
            //partition[i] and partition[i+1]. We do this so
            // partition_samples has a length of num_steps
        for (int i = 0; i < partition.size() - 1; ++i) {

                partition_samples.push_back(
                    rand_double(partition[i], partition[i+1])
                );

        }//end for

        // The integral of our function is approximately
            //the Riemann sum. We evaluate our function at the
            //random elements of partition_samples and
            //multiply this by the step size
            //between partition elements.
                //The upper and lower Riemann Sum respectively
                //take the largest and smallest value of the
                //function in an interval;
                //we randomly sample the function instead.
        for (int i = 0; i < partition_samples.size(); ++i) {
                integral_value = integral_value +
                    function(partition_samples[i]) * step_size;
        }//end for


        // Our function outputs the integral
        return integral_value;

}//end integrate_1d


// This is the function that we want to integrate
double my_function(double x) {
        return sin(x);
}// end my_function

//Program to call the integration function
int main() {

        cout << "My function integrates to ";
        cout << integrate_1d(my_function, 0.0, 1.0, 1000) << endl;

}//end main
```

1.6 Print

You've seen so far that in our simple programs we've used the *cout* command. This is an example of a Print command, which spits out text that we can read to help us to debug our code when the compiler can't catch our issues (which means that the issue is often related to faulty logic).

One area where this most often occurs is in programming For Loops, as it can often be hard to remember just how many steps a loop takes to complete, often leading to errors that often miscount by a single step, often leading to bad consequences.

2 C++

2.1 Getting Started

2.1.1 Hello World

The 'Hello World' program is a common way to get started on programming. It allows us to know the bare minimum about a language to get the program compiled and running.

Observe:

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;

    return 0;
}//end main
```

```
g++ -o hello_world hello_world.cpp
./hello_world
Hello World
```

The *include* line lets us include the *iostream* library, which defines *std::cout*. While in many physics programs we don't have to actually use the 'cout' function, it's always useful to have on hand for debugging purposes. The *return* line will return the integer 0 at the end of the program.

Note the occurence of *//end main* is a comment and does not actually compiled. Instead, we use it as a marker to denote where the *main* function ends.

2.1.2 Introductory Structure

As a general matter of structure in C++, we include a great variety of packages at the top of the file, and perhaps even state that we are using a particular namespace (to be seen later). From here we can assemble as many subprograms (called functions) as we would like, and even compile them from different files entirely.

Finally, we keep the meat of our program inside of the *main* function. We will see better implementation of this later when we build more complicated programs.

When writing code in C++, it's important for us to keep in mind that the curly braces are what matters in terms of order of execution. We may end up having a large amount of braces around one another, so it's important that we keep track of them. The comments I have added help to keep track of their purpose.

C++ does not care about the whitespace or tabs between lines of code. That's not to say that we should write our code poorly and in a bad format, but this does allow us some amount of liberty is necessary.

Indeed, we end a line in C++ code (not the text editor) with the ';' semicolon character. The compiler will read until it reaches the next semicolon and then start a new line.

For example, the following code all compiles correctly:

```
#include <iostream>

using namespace std;

int main() {
    cout << "This is clean input" << endl;

    cout <<
        "This is useful for long lines"
    << endl;

    cout << "Across" << endl; cout << "Two Lines" << endl;

    return 0;

}//end main
```

```
g++ -o lines lines.cpp
./lines
This is clean input
This is useful for long lines
Across
Two Lines
```

2.1.3 Datatypes

C++ is home to a few different types of data, and it is possible to build classes to define more, however we leave this advanced discussion until later. There are familiar integers, floats, strings, boolean variables, and so on. Moreover, there is a special *void* datatype that has no 'value' to it, which we will discuss later.

Our discussion on most of these variables have already been covered, so let's present a program that shows how to properly declare and use commonly used variable types:

```
#include <iostream>

using namespace std;

int main() {
    bool my_bool = true;

    int my_integer = 1;

    float my_float = 1.0;

    double my_double = 1.2;

    string my_string = "A string";

    return 0;

}//end main
```

We can compile this and see that it produces no errors.

There are a few other types of data, such as characters, but we suppress any mention of them for now since we won't encounter them much in computational physics.

We should be a bit careful when mixing data types when using mathematical operations. For example, observe the following program and the output:

```
#include <iostream>

using namespace std;

int main() {

    int a = 2;

    int b = 3;

    cout << "a/b = " << a/b << endl;

    return 0;

}//end main
```

```
g++ -o int_division int_division.cpp
./int_division
a/b = 0
```

Note that this result is clearly different than what we expect. This is since in C++ the division of two integers nets an integer. In particular, it gives us the floor of the rational number produced[9].

C++ contains the modulo operator % which gives us the remainder of division, itself an integer. Indeed, view this example:

```
#include <iostream>

using namespace std;

int main() {
```

[9]If we can write a/b = c$\frac{d}{b}$, we have a/b == c

```
    int a = 10;

    int b = 3;

    cout << "a/b = " << a/b << endl;
    cout << "a%b = " << a%b << endl;

    return 0;

}//end main
```

```
g++ -o more_int_division more_int_division.cpp
./more_int_division
a/b = 3
a%b = 1
```

Now, if we wanted to fix this division output and make it a rational number, we have a few ways of going about it: we can either declare *a* and *b* as floats or doubles (for which division maps to this datatype), cast the variables themselves to a new type, or we can cast the result itself to a new type. Observe the following program:

```
#include <iostream>

using namespace std;

int main() {

    int a = 10;
    int b = 3;

    //Approach 1: just declare variables that are doubles
    double c = 10.0;//Same as a
    double d = 3.0;//Same as b

    cout << "a/b = " << c/d << endl;

    //Approach 2: cast the variables as doubles
    double aa = (double)a;
    double bb = (double)b;

    cout << "a/b = " << aa/bb << endl;

    //Approach 3: cast the results as a double
    cout << "a/b = " << (double)a/b << endl;

    return 0;

}//end main
```

```
g++ -o float_division float_division.cpp
./float_division
a/b = 3.3333
a/b = 3.3333
a/b = 3.3333
```

Indeed, note how these three various methods produces the same results in the end. In order to do this, we casted variables to a different type, which just converts integers and other numbers to the desired type (and we can do this with floats and doubles to integers too). This clearly has useful applications in computations where we may want to iterate through integers, but need to use these integers to do a computation that depends on real numbers.

As it turns out, we can often divide integers and floats together and the result will automatically be casted to the correct type, but it may not be a good idea to always rely on this unless you know the inner workings of division and other mathematical operations at the language's level.

2.1.4 Conditional Statements

The primary conditional statement in C++ is the *if* loop, alongside its close partner *else*.

We input a boolean statement, and if the boolean statement is true, then the code inside the *if* statement triggers. Similarly, an *else* statement must be paired with an *if* statement, and if the *if* statement does not trigger, the *else* statement will trigger.

There is also the *else if* statement, which can come between an *if* and an *else* statement, and we can add as many of the *else if* statements in between the two.

How these statements work in C++ is that we always start withh an *if* statement, and follow it with a set of parentheses, in which we insert our boolean statement, and then follow the set of parentheses with a set of curly brackets, where we enclose the code required to execute

As it turns out, the *else if* statement is equivalent to a new *if* statement inside an *else* statement.

Observe:

```
#include <iostream>

using namespace std;

int main() {

    int my_int = 1;

    //The following two statements are equivalent:
```

```
    //1: Else if statement
    if (my_int%3 == 0) {
        cout << my_int << " is divisible by 3" << endl;
    }
    else if (my_int%3 == 1) {
        cout << my_int << "/3 has a remainder of 1" << endl;
    }//end if
    else {
        cout << my_int << "/3 has a remainder of 2" << endl;
    }//end if

    //2: Nested if statement
    if (my_int%3 == 0) {
        cout << my_int << " is divisible by 3" << endl;
    }
    else {
        if (my_int%3 == 2) {
            cout << my_int << " /3 has a remainder of 1" << endl;
        }
        else {
            cout << my_int << "/3 has a remainder of 2" << endl;
        }//end if
    }//end if

    return 0;

}//end main
```

Try and compile this code on your own. You should see that both blocks of code will produce the same output. However, it should be clear that our *else if* statement is much more compact and cleaner to read, especially so if we have more than just one *else if* statement.

Now, from here we get brought naturally to nested *if* statements. If we wanted to, we could nest these statements into one another:

```
#include <iostream>

using namespace std;

int main() {

    string my_state = "California";
    string my_county = "Riverside County";

    if (my_state == "California") {
        if (my_county == "Riverside County") {
            cout << "Your county seat is Riverside." << endl;
        }
        else if (my_county == "San Bernadino County") {
            cout << "Your county seat is San Bernadino." << endl;
        }
        else {
            cout << "I'm only programmed for Riverside ";
            cout << "and San Bernadino Counties!" << endl;
        }//end if
    }
    else if (my_state == "New Mexico") {
        if (my_county == "Los Alamos County") {
            cout << "Your county seat is Los Alamos." << endl;
        }
        else {
            cout << "I'm only programmed for ";
            cout << "Los Alamos County!" << endl;
        }//end if
    }
    else if (my_state == "Florida") {
        if (my_county == "Miami-Dade County") {
            cout << "Your county seat is Miami." << endl;
        }
        else {
            cout << "I'm only programmed for ";
            cout << "Miami-Dade County!" << endl;
        }//end if
    }
    else {
        cout << "I'm not programmed to tell you ";
        cout << "your county seat!" << endl;
    }//end if

    return 0;

}//end main
```

Next, we can use compound logic inside of these *if* statements. We can use the results to rewrite this program is an expanded form (useful for learning):

```
#include <iostream>

using namespace std;

int main() {

    string my_state = "California";
    string my_county = "Riverside County";

    if (my_state == "California" and
        my_county == "Riverside County")
    {
        cout << "Your county seat is Riverside." << endl;
    }
    else if (my_state == "California" and
        my_county == "San Bernadino County")
    {
        cout << "Your county seat is San Bernadino." << endl;
    }
    else if (my_state == "New Mexico" and
        my_county == "Los Alamos County")
    {
        cout << "Your county seat is Los Alamos." << endl;
    }
    else if (my_state == "Florida" and
        my_county == "Miami-Dade County")
    {
        cout << "Your county seat is Miami." << endl;
    }
    else {
        if (my_state == "California") {
            cout << "I'm only programmed for Riverside ";
            cout << "and San Bernadino Counties!" << endl;
        }
        else if (my_state == "New Mexico") {
            cout << "I'm only programmed for ";
            cout << "Los Alamos County!" << endl;
        }
        else if (my_state == "Florida") {
            cout << "I'm only programmed for ";
            cout << "Miami-Dade County!" << endl;
        }
        else {
            cout << "I'm not programmed to tell you ";
            cout << "your county seat!" << endl;
        }//end if
    }//end if

    return 0;

}//end main
```

Finally, we don't need to follow up an *if* statement with an *else* statement; we can in fact have a long list of *if* statements. Note though that if you follow everything with an *else* statement, that will only apply with the last *if* statement:

```
#include <iostream>

using namespace std;

int main() {

    int a = 1;

    if (a%2 == 1) {
        cout << a << " is not divisible by 2." << endl;
    }//end if
    if (a == 1) {
        cout << a << " is neither prime nor composite." << endl;
    }//end if
    if (a == 3) {
        cout << a << " is the smallest odd prime number." << endl;
    }
    else {
        cout << a << " is not 3." << endl;
    }//end if

    return 0;

}//end main
```

```
g++ -o onlyifs onlyifs.cpp
./onlyifs
1 is not divisible by 2.
1 is neither prime nor composite.
1 is not 3.
```

2.1.5 Loops

Loops are powerful tools. They allow us to execute again and again without having to copy-paste it all over the place. This saves space in documents and makes code easier to read. The downside is that sometimes loops can be a bit complicated, and it can be easy to make an infinite loop that never terminates.

Logically, we start with the first condition, and we then have a statement that we compare to; if the second statement is true, then the loop fires. We then update our first condition with the iterator.

In particular, we can use loops to numerically calculate integrals, sums, derivatives, and other interesting quantities, alongside other fun things. But before we learn about these applications, let's learn the basics.

For Loops For Loops are loops that have three inputs: an initial condition, a final condition, and an iterator. We have a variable that starts at an initial condition, and it iterates until it reaches the final condition. Its progression from the initial to the final values are dictated by the iterator.

Observe when we simplely iterate by an integer

```
#include <iostream>

using namespace std;

int main() {

    for (int i = 0; i < 4; ++i) {
        cout << "i = " << i << endl;
    }//end for

    return 0;

}//end main
```

```
g++ -o forloop_ex1 forloop_ex1.cpp
./forloop_ex1
i = 0
i = 1
i = 2
i = 3
```

We also need not just advance by one, but we can do so by 2 or any integer amount (including decreasing).

```
#include <iostream>

using namespace std;

int main() {

    for (int j = 0; j < 6; j = j + 2) {
        cout << "j = " << j << endl;
    }//end for

    return 0;

}//end main
```

```
g++ -o forloop_ex2 forloop_ex2.cpp
./forloop_ex2
j = 0
j = 2
j = 4
```

We also don't need to iterate via addition; we are also free to use multiplication, or indeed any nice operation[10].

```
#include <iostream>

using namespace std;

int main() {

    for (int k = 1; k < 10; k = 2*k) {
        cout << "k = " << k << endl;
    }//end for

    return 0;

}//end main
```

```
g++ -o forloop_ex3 forloop_ex3.cpp
./forloop_ex3
k = 1
k = 2
k = 4
k = 8
```

We also don't need to be restricted to advancing through integers, but we are free to use other datatypes, such as doubles.

```
#include <iostream>

using namespace std;

int main() {

    for (double w = 0.0; w < 0.5; w = w + 0.1) {
        cout << "w = " << w << endl;
    }//end for

```

[10]Note that this loop requires the initial condition of $k = 1$ or else we get an infinite loop.

```
    return 0;

}//end main
```

```
g++ -o forloop_ex4 forloop_ex4.cpp
./forloop_ex4
w = 0
w = 0.1
w = 0.2
w = 0.3
w = 0.4
```

Note that we are more or less free to execute arbitrary code inside the *for* loop. We are able to set *if* statements inside, set in more *for* loops, do continual mathematical computations, and so on.

For loops are extremely powerful and a near necessity for doing mathematical computations given the intrinsically mathematical nature of the loop. We will see applications of this later on.

While Loops While *For* loops usually calculate things a set number of times, iterated through in an often mathematical matter, *While* loops are strictly logically based. If the condition of a *While* loop is true, then the loop will continue to execute.

It should be clear that we can emulate a *for* loop with a *while* loop and vice versa, but as it turns out, *while* loops are just better for strictly logical applications.

Observe such an example:

```
#include <iostream>

using namespace std;

int main() {

    bool i_want_to_leave = false;

    string my_answer;

    while (i_want_to_leave == false) {
        cout << "Do you want to leave the loop? ";
        cin >> my_answer;

        if (my_answer == "Yes" or my_answer == "yes") {
            i_want_to_leave = true;
            cout << "Let's leave the loop." << endl;
        }//end if

    }//end while

    return 0;

}//end main
```

Note that this program involves the *cin* line of code that we haven't really discussed. Suffice to say that it accepts user input.

```
g++ -o wanttoleave wanttoleave.cpp
./wanttoleave
Do you want to leave the loop? No
Do you want to leave the loop? a
Do you want to leave the loop? :)
Do you want to leave the loop? Yes
Let's leave the loop.
```

Basically, in this program, we continue to execute the code inside the loop, asking if we want to leave the loop. After getting user input through *cin*, we check to see if the user input is 'Yes' or its uncapitalized version. If it is, we set a flag to leave the loop and then print a message before the loop stops running.

For the purposes of simple enough physics computations, we will not need *while* loops very often, so we limit their discussion. Needless to say, unlike *for* loops, there isn't much to go over at a simple level.

Do While Loops A *do while* loop is another type of loop. Much like the other two loops we have gone over, it more or less shares an equivalent functionality and its convenience is purely notational[11].

A *do while* loop executes a block of code before checking the condition that determines whether or not it terminates. This means that such a loop will always execute code once, and depending on other factors (at our control), it may execute more times.

See this example on how it works:

```
#include <iostream>

using namespace std;

int main() {

    bool i_want_to_stay = false;

    string my_answer;

    do {
        cout << "Do you want to stay in the loop? ";
```

[11] Albeit I see no reason on why to use these loops.

```
        cin >> my_answer;

        if (my_answer == "Yes" or my_answer == "yes") {
            i_want_to_stay = true;
        }
        else if (my_answer == "No" or my_answer == "no") {
            i_want_to_stay = false;
            cout << "Let's leave." << endl;
        }//end if

    }//end do
    while (i_want_to_stay == true);

    return 0;

}//end main
```

```
g++ -o doloop doloop.cpp
./doloop
Do you want to stay in the loop? Maybe

./doloop
Do you want to stay in the loop? No
Let's leave.

./doloop
Do you want to stay in the loop? Yes
Do you want to stay in the loop? Maybe
Do you want to stay in the loop? No
Let's leave.
```

Note how in the first instance of running the program, entering a string that was neither 'Yes' nor 'No' meant that the program (initialized with our variable being false) terminated the loop since the condition was no longer valid. Similarly, answering 'No' in a new instance, as a first answer, let us leave. However, if we answer 'Yes', the loop has the condition to continue executing beyond the first time, and from then on a non-answer to the question continues the loop.

Again, we will rarely use *do while* loops, so the discussion that we need on them will be minimal.

Nesting Loops Together We are free to nest loops inside one another. This is an incredibly useful tool to have at our disposal, as we cannot easily describe certain things (like multivariable functions) with just one index, and often times even in the case that we can do this, it may just be simpler to use nested loops.

There is a limit to just how many loops we can nest inside one another, however, this limit is sufficiently large (at least 256) that we don't need to worry about this limit.

So, look at this simple example that shows one application of nested loops:

```
#include <iostream>

using namespace std;

int main() {

    for (int i = 0; i < 4; ++i) {

        for (int j = 0; j < 4; ++j) {

            cout << i << " * " << j << " = " << i*j << endl;

        }//end for

    }//end for

    return 0;

}//end main
```

```
g++ -o nestloop nestloop.cpp
./nestloop
0 * 0 = 0
0 * 1 = 0
0 * 2 = 0
0 * 3 = 0
1 * 0 = 0
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
3 * 0 = 0
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
```

Note how that we are able to access both indices of the loop in the innermost loop. This applies to not just two nested loops, but we would be free to create larger and larger nested loops to produce similar multiplication tables to the one given. We can also do more with it, but that will all come later.

Break Sometimes we need to break from a loop before completing the requirements for exit[12]. For this purpose, we have *break* which, upon being called breaks us out of the first loop that we are in. This applies for any type of loop. Indeed, feel free to use this functionality, as it is quite efficient in terms of time, and it can often be faster to leave a *while* loop via break than through making its boolean value false.

Anyways, let's look at an explicit example where a *break* statement is important for the functionality of a simple program.

```
#include <iostream>

using namespace std;

int main() {

    bool i_am_a_foo = false;

    string my_type;
    string am_i_funny;

    while (i_am_a_foo == false) {
        cout << "Are you a Foo or a Bar?" << endl;
        cout << "I am a ";
        cin >> my_type;

        if (my_type == "Foo") {
            cout << "You are a foo." << endl;
            i_am_a_foo = true;
        }
        else if (my_type == "Bar") {
            cout << "You are a Bar." << endl;
            break;
        }
        else {
            cout << "Try again." << endl;
        }//end if
    }//end while

    while (i_am_a_foo == true) {
        cout << "Are you a funny Foo? ";
        cin >> am_i_funny;
```

[12]Or we may just use a *break* to exit an otherwise infinite loop

```
        if (am_i_funny == "Yes") {
            cout << "You are funny." << endl;
            break;
        }
        else if (am_i_funny == "No") {
            cout << "You are not funny." << endl;
            break;
        }
        else {
            cout << "Try again." << endl;
        }//end if
    }//end while

    return 0;

}//end main
```

```
g++ -o foo_or_bar foo_or_bar.cpp
./foo_or_bar
Are you a Foo or a Bar?
I am a Giant
Try again.
Are you a Foo or a Bar?
I am a Bar
You are a Bar.

./foo_or_bar
Are you a Foo or a Bar?
I am a Foo
You are a Foo.
Are you a funny Foo?
a
Try again.
Yes
You are funny.
```

Note how in this particular example that we used a *while* loop (with a convenient initial condition) to get the answer to a question. The loop allowed us to keep on asking the question until we got a desired answer. If we were a Foo, the related boolean would be set as true, and this would break us out of the loop. Similarly, if we were a Bar, we were not a Foo (assuming they were mutually exclusive). We had answered the question, but the *while* loop would be satisfied by being a Foo, which we were not, so we broke the loop instead.

The importance of whether or not we were a Foo came into fruition in the next *while* loop. If we were a Foo, it would trigger and ask us if we were funny. If we were a Bar, the section of code would not trigger, but if we were a Foo, we would get a similarly designed section of code where we answer whether or not we were funny. From here the statement breaks again due to the fact our loop was controlled by our positive Foo-ness.

This is most certainly a bit of an abstract example, but hopefully you see that it has concrete applications (perhaps not so much in physics). Perhaps a Foo and a Bar are 2 DND races, and a Bar does not get any bonus to being funny, but if a Foo is funny they get a bonus to their Charisma.

Continue The *continue* statement lets us break a statement in a loop for a single iteration. Basically, once the computer sees *continue*, it immediately jumps to the next iteration in the loop.

For example:

```
#include <iostream>

using namespace std;

int main() {

    for (int i = 2; i < 7; ++i) {

        if (i == 4) {
            cout << "I did not do any push-ups." << endl;
            continue;
        }//end if

        cout << "I did " << i << " push-ups in a row." << endl;

    }//end for

    return 0;

}//end main
```

```
g++ -o continue continue.cpp
./continue
I did 2 push-ups in a row.
I did 3 push-ups in a row.
I did not do any push-ups.
I did 5 push-ups in a row.
I did 6 push-ups in a row.
```

Now, hopefully you can see the fact that while that was nice and all, we could have done the exact same thing by instead using an *else* statement. This is true. However, no doubt that you may be able to find an excuse to use *continue* in some situation, such as to prevent there from being a large amount of *if-else* statements within a loop.

2.1.6 Functions

In C++, we are able to define functions. This is very general thing, but how a function works is that we declare the function, with the datatype that it is supposed to return. The function must be declared with all of its arguments that it will use.

Non-void Functions We can declare a function of any datatype, including a special void type, which has no return value. As a nice rule of thumb, functions that return real numbers, integers, and other types of numbers can be used precisely as familiar mathematical functions.

Indeed, observe the following example:

```
#include <iostream>

using namespace std;

double times_two(double x) {
    return 2.0 * x;
}//end times_two

int main() {

    double x = 1.0;
    cout << "x = " << x << endl;
    cout << "2.0 * x = " << times_two(x) << endl;

    x = 2.0;
    cout << "x = " << x << endl;
    cout << "2.0 * x = " << times_two(x) << endl;

    x = 3.0;
    cout << "x = " << x << endl;
    cout << "2.0 * x = " << times_two(x) << endl;

    return 0;

}//end main
```

```
g++ -o times_two times_two.cpp
./times_two
x = 1.0
2.0 * x = 2
x = 2.0
2.0 * x = 4
x = 3.0
2.0 * x = 6
```

Similarly, functions can accept multiple arguments:

```
#include <iostream>

using namespace std;

double product(double x, double y) {
    return x*y;
}//end product

int main() {

    double x = 3.0;
    double y = 4.0;

    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    cout << "x*y = " << product(x,y) << endl;

    return 0;

}//end main
```

```
g++ -o xyprod xyprod.cpp
./xyprod
x = 3.0
y = 4.0
x*y = 12
```

Indeed, this example of a function is incredibly basic, but it should be clear that we can write more and more complicated functions. For example, here's the sign function, which returns the sign of the double input:

```
#include <iostream>

using namespace std;

double sgn(double x) {
    if (x < 0.0) {
        return -1.0;
    }
    else if (x == 0.0) {
        return 0.0;
    }
    else {
        return 1.0;
    }//end if
}//end sgn

int main() {

    double x = -3.0;
    cout << "x = " << x << endl;
    cout << "sgn(x) = " << sgn(x) << endl;

    x = 0.0;
    cout << "x = " << x << endl;
    cout << "sgn(x) = " << sgn(x) << endl;

    x = 4.0;
    cout << "x = " << x << endl;
    cout << "sgn(x) = " << sgn(x) << endl;

    return 0;

}//end main
```

```
g++ -o sgn sgn.cpp
./sgn
x = -3.0
sgn(x) = -1
x = 0.0
sgn(x) = 0
x = 4.0
sgn(x) = 1
```

Now, we can even make a factorial calculator with a function:

```
#include <iostream>

using namespace std;

int for_factorial(int n) {

    int fact_n = 1;

    if (n < 0) {
        return 0;
    }//end if
    if (x == 0) {
        return 1;
    }//end if
    for (int i = n; i > n; --i) {
        fact_n = fact_n * i;
    }//end for

    return fact_n;

}//end for_factorial

int main() {

    for (int n = 0; n < 6; ++n) {
        cout << "n = " << n << endl;
        cout<< "n! = " << for_factorial(n) << endl;
    }//end for

    return 0;

}//end main
```

```
g++ -o for_factorial for_factorial.cpp
./for_factorial
n = 0
n! = 1
n = 1
n! = 1
n = 2
n! = 2
n = 3
n! = 6
n = 4
n! = 24
n = 5
n! = 120
```

As it turns out, functions can be called recursively, so we can call a function within itself. This is best demonstrated with an example. Let's revisit the factorial function, but call it recursively:

```
#include <iostream>

using namespace std;

int recur_factorial(int n) {

    if (n < 0) {
        return 0;
    }//end if
    if (n == 0) {
        return 1;
    }
    else {
        return n * recur_factorial(n-1);
    }//end if

}//end recur_factorial

int main() {

    for (int n = 0; n < 6; ++n) {
        cout << "n = " << n << endl;
        cout<< "n! = " << recur_factorial(n) << endl;
    }//end for

    return 0;

}//end main
```

```
g++ -o recur_factorial recur_factorial.cpp
./recur_factorial
n = 0
n! = 1
n = 1
n! = 1
n = 2
n! = 2
n = 3
n! = 6
n = 4
n! = 24
n = 5
n! = 120
```

Void Functions Functions of a *void* type behave a little bit differently than normal functions. Really, not much is different other than the fact that they don't have a return value. We can't return anything from them. However, we are still free to terminate them with the *return* statement, but we just can't have a variable be returned.

This does seem to be of some limited usefulness at first glance, but ultimately, *void* functions allow us to write blocks of code that we can call at will.

See this example that prints *Hello World!*:

```
#include <iostream>

using namespace std;

void hello_world_print() {

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

    return;

}//end hello_world_print

int main() {

    for (int i = 0; n < 4; ++i) {
        hello_world_print();
    }//end for

    return 0;

}//end main
```

```
g++ -o hello_world_print hello_world_print.cpp
./hello_world_print
Hello World!
Hello World!
Hello World!
Hello World!
```

But don't be mistaken, a *void* function is perfectly capable of accepting arguments (and even multiple of them). Observe:

```
#include <iostream>

using namespace std;

void string_print(string my_string) {

    cout << my_string << endl;

    return;

}//end string_print

int main() {
    string my_string = "Hi";

    for (int i = 0; n < 4; ++i) {
        string_print(my_string);
    }//end for

    return 0;

}//end main
```

```
g++ -o string_print string_print.cpp
./string_print
Hi
Hi
Hi
Hi
```

Passing by Reference As it turns out, when a function *f*, which depends on a variable a is called, the computer makes a copy of the variable a and then uses that copy in the function. It implicitly assumes that a itself is not modified. Moreover, this aspect of copying variables has implications on the speed of execution of the function.

However, we can pass a variable by reference using the ampersand '&', which denotes that we modify the variable in the function (alongside any other return values it computes). This is a faster, albeit perhaps a slightly unsafe way of doing computations.

Observe this example that passes two numbers by reference and computes their square:

```
#include <iostream>

using namespace std;

void my_product(double& x, double& y) {

    x = x*x;
    y = y*y;

}//end my_product

int main() {

    double x = 2.0;
    double y = 3.0;

    cout << "x = " << x << endl;
    cout << "y = " << y << endl;

    my_product(x,y);

    cout << "x = " << x << endl;
    cout << "y = " << y << endl;

    return 0;

}//end main
```

```
g++ -o prod_ref prod_ref.cpp
./prod_ref
x = 2
y = 3
x = 4
y = 9
```

3 Python

4 Fortran90

5 Mathematica