Introduction to C++: I

C++ is the programming language that will be used throughout the computational classes. Whilst you can do some quite complex operations in bash, it is not suitable for scientific programming for a variety of reasons. Languages such as C, C++ and Fortran are widely used in research. Some future research projects might be in Fortran but the much of what you will learn about in C++ will be immediately applicable.

Resources

There is a wealth of information available on the internet.

http://www.cprogramming.com/

http://www.cplusplus.com/

I also like Accelerated C++ by Koenig and Moo.

http://c.learncodethehardway.org/book/ is an excellent introduction to C (but most of it applies to C++ as well).

What isn't covered

C++ is a much larger language than C. In particular we will not cover the object-oriented, template and string manipulation facilities and provided by C++. In fact (for those familiar with C) the main difference between the C++ shown here and standard C is the way memory, pointers and arrays are handled.

Hello world

As an example:

```
// A short C++ program to greet the world.
#include <iostream>

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

Compilation

C++ is, unlike bash (and other interpreted languages such as perl and python), a compiled language. An interpreted language to interpreted into the necessary machine-specific instructions at runtime. A compiled language is has to be reduced to the machine-specific instructions before it can be run. This makes interpreted languages easier to develop (as they don't need to be recompiled in order to test a new section of code) and more flexible, but at the cost of efficiency. Compiled languages are substantially faster than interpreted languages (in general). This trade-off results in a balancing act: for some applications are easier to do in certain languages than others. Simulations of materials are much more demanding than normal computer tasks. Almost all scientific programs are written in compiled languages for this reason.

A free (as in beer and speech) C++ compiler is provided by the GNU GCC suite (gcc.gnu.org).

The speed of a compiled executable depends upon the optimisation flags and the compiler used. Some compilers---particularly commercial ones---produce substantiall more efficient executables than g++. For certain tasks using a *good* compiler and optimisation settings can increase the speed of a program by more than a factor of 2. However, we shall not concern ourselves with either set of options now. g++ is a perfectly adequate compiler for our purporses.

To compile a single C++ source file:

```
$ g++ source_code.cpp -o program_name
```

To emphasise the distinction between C++ code and bash commands, I have using the $ to indicate the bash prompt (it should not be typed!).

As with shell scripts, it is conventional (but not necessary) to use a specific filename extension to indicate the file type: for C++ the .cpp ending is common. So to compile our hello world program:

```
$ g++ hello.cpp -o hello
```

Execution

g++ gives a file that is already executable. We can run it in the same way we ran our shell scripts:

```
$ ./hello
```

Hello world: reprise

We shall now examine the hello world program in sections.

Comments

```
// A short C++ program to greet the world.

```

Anything after a // in a line is treated as a comment. Multi-line comments can be made either by using // repeatedly or by using C-style comments, "/* *and* */":

```
/*Everything between the two
   markers is treated as a comment*/
```

Pre-processing

```
#include <iostream>
```

C++ distinguishes between features which are part of the **core** language (and so are always available) and features which are part of the standard library (and so must be specifically requested). Lines beginning with a # are (pre-)processed before the rest of the code is compiled. The include directive causes the required **header** file to be included. A header file contains the declarations of the functions are variables implemented in the associated library. The angle brackets indicate iostream is part of the standard library (which must be provided by any C++ compiler). We do not need to concern ourselves with details about how the standard library is included nor how it's implemented.

The standard library guarantees the *portable* behaviour of various functions across platforms: unix, linux, windows etc.

main

```
int main()
```

A function is a section of code which has a name and so can be run by calling that name.

Every C++ program *must* have a main function: it is the function that is run when the program is executed.

We shall discuss functions in more depth later. A function can return a value of a certain type. Here we declare that the main function will return an integer. main is a special function that **must** return an integer: this is required by the C++ standard.

Braces

```
{
}
```

Curly braces are used to enclose a discrete unit of code. The unit of code might enclose a function---as it does here---or a logical section or just a block of code. If the braces enclose two or more lines then they are executed sequentially.

Writing output

```
std::cout << "Hello world!" << std::endl;
```

A C++ statement (i.e. anything that does not start or end a code block or is a # directive) is terminated by a semi-colon. More than one statement can be placed on the same line but at the price of making the code harder to read. This line causes the "Hello world!" to be printed to the standard output stream followed by a new line. We shall discuss it more in a minute.

C++ uses **namespaces**. A namespace is used to collect together a group of related variables and functions and helps avoid the clashes in the names of very different items. The namespace an item is in is indicated by using *namespace_name::item_name*. Hence *std::cout* refers to *cout* in the *std* namespace. The standard library uses the *std* namespace throughout. Writing *std::* in front of everything from the standard library becomes tedious. Fortunately there's a way to "import" a namespace into the current scope:

```
using namespace std;
```

This means that *std::* no longer has to prefix items from the *std* namespace. We will use this throughout the course.

Returning a value

```
return 0;
```

main declared that it would return an integer. This statement returns a 0. The return statement causes execution of the current function to end and the value between return and the semi-colon to be passed back to whatever called the function.

Remember the && and || operators in bash? They depend upon the exit code of the previous command. In C++ this exit code is given by the value returned from the main function and allows the user to determine if the program executed successfully or not.

Reprise

Piecing it back together (and using the standard namespace):

```
// A short C++ program to greet the world.
#include <iostream>
using namespace std;
```

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

Introduction to C++: II

Variables

C++ is a statically typed language. This means that a variable must be declared to be of a certain kind before it is used. (Compare this to bash, where as variable could be used without such restrictions: bash is an example of a dynamically typed language.) Furthermore operations can behave differently with different types or work only with certain types. Note that C++ is **case sensitive**.

Fundemental types are:

Name	Description	Size (bytes)	Range
char	A single character (or small integer)	1	signed: -128 to 127 unsigned: 0 to 255
short int	A short integer	2	signed: -32768 to 32767 unsigned: 0 to 65535
int	An integer	4	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. Two values: true or false	1	True or false
float	A single-precision floating point number	4	+/- 3.4 e +/-38
double	A double-precision floating point number	8	+/- 1.7 e +/-308

In Dr. Haynes' lectures you will learn why the size and range (and if it's signed or unsigned for an integer) of a data item of a specific type are intrinsically linked.

Another useful type is the string type. It is defined in the string section of the standard library.

To use a variable we must declare what type it is and give the variable a name:

```
int a;
std::string hi;
```

A variable can then be assigned using the '=' operator:

```
int a;
std::string hi;
a = 42;
hi = "hello";
```

As C++ allows a variable to be declared and assigned in one line, these statements can be written compactly as:

```
int a = 42;
std::string hi = "hello";
```

Multiple variables of the same type can be declared on the same line:

```
int a = 42, b, c = 13;
```

The "hello world" program could thus be written as:

```
// A short C++ program to greet the world.
#include <iostream>
using namespace std;

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

(NB: technically the {} have an impact on a variable's scope: a fact we shall not expound upon here, but you need to take care when a variable is not declared at the start of a function.)

Input and output

```
cout << "Hello world!" << endl;
```

C++ uses data **streams** for input and output[1]. The operator << sends data to an output destination. Repeated uses of << appends items to the stream. *cout* is the output defined for the standard output (we need not worry about how it's implemented under different operating systems). We equally could have written:

```
cout << "Hello" << " " << "world" << "!" << endl;
```

endl is a constant which defines the end of a line.

In a similar fashion input from standard input can be obtained:

```
// Ask for and print the user's age.
#include <iostream>
using namespace std;

int main() {
    int age;
    cout << "Please enter your age and press return: ";
    cin >> age;
    cout << "You are " << age << "years old" << endl;
    return 0;
}
```

Note the use of the >> operator for the input. *cin* is also defined in the standard library and is the standard input.

A brief comment on style

C++ does not take whitespace into account. As far as the compiler is concerned, the "hello world" program given above is identical to:

```
#include <iostream>
using namespace std; int main() { cout << "Hello world!" << endl;}
```

However the lack of comments, spaces, newlines and identation make it harder to read. Imagine if it had been a program that was hundreds or thousands of lines long! Whilst there is more to programming than how the source code is formatted, it's not a bad place to start and greatly aids the maintainance of a program. Try to have the layout and indentation reflect the logical structure of the program, comment frequently and pick meaningful variable names: code is read far more than it is written.

Establishing good habits now will pay off when some of your code needs to be modified by you or a colleague months or years after it was first written.

Maths functions

The operators +, -, * and / perform addition, subtraction, multiplication and division as expected.

However, care needs to be taken with division involving integers:

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

int main() {
    cout << 3./4 << endl;
    cout << 3/4. << endl;
    cout << 3/4 << endl;
    return 0;
}
```

Other mathematical functions such as exponentiation, logarithms, exponentials and square roots are defined in the standard library and are used by including the cmath header file.

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

int main () {
    cout << pow(2.4,2) << endl; // 2.4^2
    cout << exp(2.4) << endl;   // e^2.4
    cout << log(2.4) << endl;   // log(2.4) (natural log)
    cout << sqrt(2.4) << endl;  // 2.4^0.5
    cout << log10(2.4) << endl; // log10(2.4) (log base 10)
}
```

Care needs to be taken as these functions are not defined for integer arguments (apart from exponentiation where integer powers are allowed). They are defined for floats and doubles, however.

If the function is unambigous then the integer is implicitly converted to a double (thus avoiding loss of precision). The main time this causes a problem is when raising an integer to an integer power: there are multiple possible functions that could be called. The solution is to **explicitly** convert the integer into a double (or a float, as desired). Converting a variable type is called typecasting and can be done in two ways:

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

int main() {
    // cout << pow(2,2) << endl; // Gives a compiler error as int^int is not defined.
    cout << pow( double(2), 2) << endl;  // functional method
    cout << pow( (double) 2, 2) < endl;  // C method
}
```

Both ways are equivalent. In general you can do either of:

```
new_type (variable);
(new_type) variable;
```

Conditions

As with bash, it is possible to test whether a statement is true or false in C++.

!x

Logical negation. If x is true, then !x is false.

x == y

Returns true if x is equal to y.

x != y

Returns true if x is not equal to y.

x && y

Returns true if both x and y are true. y is only evaluated if x is true.

x || y

Returns true if x or y are true. y is only evaluated if x is false.

These can be used in if-blocks and for loops (see later). The if block syntax is:

```
if (condition1) {
    // do stuff if condition1 is true.
} else if (condition2) {
    // do stuff if condition2 is true.
} else if (condition2) {
    // do stuff if condition3 is true.
} else {
    // do stuff if condition1, condition2 and condition3 are all false.
}
```

The *else if* and *else* blocks are optional. For example:

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

int main() {
    int x;
    cout << "Enter an integer: ";
    cin >> x;
    if (x < 100) {
        cout << "x is less than 100" << endl;
    } else if (x > 100) {
```

```
        cout << "x is greater than 100" << endl;
    } else if (x == 100) {
        cout << "x is equal to 100" << endl;
    }
    return 0;
}
```

Multiple conditions can be tested at once:

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

int main() {
    int x, logx;
    cout << "Enter an integer: ";
    cin >> x;
    logx = log(x);
    if ( (x < 10) && (logx < 10) {
        cout << "x and log(x) are less than 10." << endl;
    }
    return 0;
}
```

Tasks

1. Create a directory in your subversion repository to contain the work from this session. Check out a working copy of the repository.
2. Test the "hello world" program.
3. Write a "hello me" program that asks for the user's name and then greets the user.
4. Write a program which does the following:

    a. Reads a number in from standard input.

    b. Squares it.

    c. Takes log10 of the result.

    d. Multiples the result by 12.

    e. Raises e to the power of the result.

    f. Takes the square root.

    Print out the value at each stage. Use a separate (new) directory in your subversion repository for this program.

Introduction to C++: III

Loops

Conditions can be used to loop over the same operations until a condition is satisfied. We shall look at two types of loop: *while* and *for*.

while

The *while* loop syntax is:

```
while (condition) {
    // Statements to do whilst the condition is true.
}
```

For example:

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

int main() {
    string ans;
    cout << "Enter y or n followed by RETURN: ";
    cin >> ans;
    while ( ans != "y" && ans != "n" ) {
        cout << "You didn't enter y or n.  Enter y or n followed by RETURN: ";
        cin >> ans;
    }
    if (ans=="y") {
        cout << "You entered y." << endl;
    } else {
        cout << "You entered n." << endl;
    }
    return 0;
}
```

This causes the user to be asked to enter a response until they reply with either "y" or "n".

Another example:

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

int main() {
    int x = 0;

    // Count from 0 to 10.
    while (x <= 10) {
       cout << x << endl;
       x++; // x++ is equivalent to x = x + 1;
            // There is a analagous -- operator.
    }

    cout << "Done counting." << endl;
    return 0;
}
```

The while loop tests that x is less than or equal to 10. If it is then the statements within the while block (deliminated by the braces) are executed in order. When x is no longer equal to 10 the loop is exited.

while loops are often used to loop over an integer range. *for* loops can do the same thing and adjust the value of the iteration accordingly. They take the form:

```
for (initialisation_statement; condition; expression) {
    // do stuff
}
```

for loops start with initialising a variable. At the end of each cycle through the for loop the provided expression is applied. The loop ends when the condition is no longer met. For instance the *while* loop above could be written as:

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

int main() {
    int x;

    for (x = 0; x <=10 ; x++) {
        cout << x << endl;
    }

    cout << "Done counting." << endl;
    return 0;
}
```

Functions

Up until now we've written everything in *main*. But what if we want to do two similar things? Further, having all code within *main* quickly becomes unwieldly as the length of the program grows. We can split up the program into sections called functions. This allows for re-use of code and makes programs substantially easier to modify and maintain.

Compare:

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    int i;
    for (i=0;i!=10;i++) {
        cout << i << " " << pow(i,2.0) << endl;
    }
    cout << "Repeating loop." << endl;
    for (i=0;i!=10;i++) {
        cout << i << " " << pow(i,2.0) << endl;
    }
    return 0;
}
```

to:

```
#include <iostream>
#include <cmath>

using namespace std;
```

```
void loop_over_10() {
    for (i=0;i!=10;i++) {
        cout << i << " " << pow(i,2.0) << endl;
    }
}

int main() {
    int i;
    loop_over_10();
    cout << "Repeating loop." << endl;
    loop_over_10();
    return 0;
}
```

The *void* keyword indicates that the *loop_over_10* function does not return any value. The *loop_over_10* function can be used like a normal statement.

Functions can return values and be used in expressions (as *main* is):

```
#include <iostream>
#include <cmath>

double quadratic_fn(double x) {
    return pow(x,2)+2*x+2;
}

int main() {
    cout << quadratic_fn(2.0) << endl;
    cout << quadratic_fn(4.562) << endl;
    return 0;
}
```

Arguments

The previous example contained the use of an argument in a function. This enables input to be passed into the function. Note that the type of the arguments must be declared in the argument list. C++ passes arguments by **value** by default. This means that the variables are copied and are local to the function:

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

double modify_x(double x) {
    x=x*2;
    return x;
}

int main() {
    double x = 10.123;
    cout << x << endl;
    cout << modify_x(x) << endl;
    cout << x << endl;
    return 0;
}
```

We shall discuss this more in the next session.

Finally *main* can take arguments from the command line. Arguments are stored as character arrays and so need to be converted to integers and floats/doubles using the *atoi* and *atof* functions as appropriate.

```
#include <iostream>
#include <cstdlib> // provides atoi and atof.
using namespace std;

int main(int argc, char** argv) { // char** will be discussed in the next session.
    // Example usage: test_main_args 8 10.234
    int i, j;
    double x;
    cout << "Number of arguments: " << argc << endl;
    for (i=1;i<argc;i++) {
        cout << "argument " << i << " = " << argv[i] << endl;
    }
    j = atoi(argv[1]); // convert argument to integer.
    x = atof(argv[2]); // convert argument to double.
    cout << j << "   " << j*3 << endl;
    cout << x << "   " << x*3 << endl;
    return 0;
}
```

Tasks

5. In a new directory in your subversion repository write a program which reads in two integers and prints out appropriate statements if both are less than 50, both greater than 50 or one is less than 50 and one greater than 50.
6. Write a program which prints out odd numbers less than 20.
7. One way to estimate the value of π is to pick random points in a unit square, centred on the origin, and count the number of points which fall within the cirle enscribed within the square. π can be determined from the ratio of the number of points within the circle to the total number of points.

   Write a program which will use do this using a number of points given by the user on the command line. Use a new directory (called estimate_pi) in your subversion repository for this task. Complete before the next session.

Notes

A simple random number generator is provided by cmath in the standard library. *rand()* returns a random integer between 0 and a constant called *RAND_MAX* (which is also defined in cmath). Thus a random number between 0 and 1 can be obtained:

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

int main() {
    cout << double(rand())/RAND_MAX << endl;
    cout << double(rand())/RAND_MAX << endl;
    cout << double(rand())/RAND_MAX << endl;
    cout << double(rand())/RAND_MAX << endl;
    return 0;
}
```

Repeated calls to *rand()* produces a sequence of random numbers. The implementation of a good random number generator is decidedly non-trivial. For calculations which are dependent upon large numbers of

random numbers (such as Monte Carlo simulations) the random number generator in the standard library is **not** adequate.

Introduction to C++: IV

Pointers

A variable (or, more specifically, its value) is stored in a particular point in memory. A fundemental concept in C and C++ is that another variable (called a **pointer**) can represent the **address** of another variable. There are two operators which are used to manipulate pointers:

address operator, &

The address operator returns the address of a variable---i.e. the place in the memory used to store the value of that variable. Thus &x is the address of the object x.

dereference operator, *

The dereference operator returns the variable which the pointer points to. Thus if p is the address of x, *p gives the value of x.

```
#include <iostream>

using namespace std;

int main() {
    int x;
    int *p;

    x = 42;
    p = &x;
    cout << "Value of x: " << x << endl;
    cout << "Address of x: " << &x << endl;
    cout << "Pointer p: " << p << endl;
    cout << "Dereferencing p: " << *p << endl;
    return 0;
}
```

p is a **pointer** and points to x. Pointers are declared according to the datatype which they point to and so *p is declared to be type int as pointer p gives an integer when it is dereferenced.

Operations can be performed using both x and p. Care must be taken however to dereference p correctly:

```
#include <iostream>

using namespace std;

int main() {
    int x;
    int *p;

    x = 42;
    p = &x;
    cout << "Value of x: " << x << endl;
    cout << "Pointer p: " << p << endl;
    cout << "Dereferencing p: " << * p << endl;
    x += 1; // x is now 43
    cout << "Value of x: " << x << endl;
```

```
    cout << "Dereferencing p: " << * p << endl;
    *p += 1; // x is now 44
    cout << "Value of x: " << x << endl;
    cout << "Dereferencing p: " << * p << endl;
    // THIS CHANGES WHERE p POINTS TO AND MAY CAUSE A SEGMENTATION FAULT!
    p += 1;
    cout << "Value of x: " << x << endl;
    cout << "Pointer p: " << p << endl;
    cout << "Dereferencing p: " << *p << endl;
    return 0;
}
```

Static arrays

So far we have only used single variables. This is a severe restriction in many cases.

An array is a container of variables that is part of the core language. For instance an array containing 3 elements is declared using:

```
double vec[3];
```

Individual elements are references using square brackets:

```
for (i=0;i<=2;i++) {
    vec[i] = 2*i+1;
}
```

Note that array indexing in C and C++ starts from 0.

There is a close relationship between pointers and arrays. Let's explore:

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

int main() {
    double coords[3];
    for (int i=0;i<=2;i++) {
        coords[i] = 2*i+1;
        cout << "Component " << i << " of coords: " << coords[i] <<  endl;
    }
    cout << "coords " << coords << endl;
    cout << "&coords[0] " << &coords[0] << endl; // same as coords!
    return 0;
}
```

A variable array is actually just the first pointer in a consecutive sequence of pointers. Thus *coords[n]* is equivalent to **(coords+n)*. *coords+1* is the address of element number 1 in the coords array, *coords+2* the address of element number 2 and so on.

Array indexing is fundemental to most languages. In C and C++ it is not a direct property of arrays, but rather a result of the behaviour of pointers and array names.

Arguments to main

We are now in a position to understand the arguments to the *main* function.

C and C++ provide a way for command line options to be given to a program. We can write programs which accept command line options by specifying two arguments to the *main* function, often (but not necessarily) called argc and argv:

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

int main(int argc, char **argv) {
    // Compile using (eg) g++ test_args.cpp -o test_args
    // Run using (eg)
    // ./test_args hello world
    // ./test_args this is a test
    // ./test_args 1 1 2 3 5 8 13 21 34 55
    int i;
    cout << "Number of arguments given on command line: " << argc << endl;
    for (i=1;i<argc;i++) {
        cout << "argument " << i << " = " << argv[i] << endl;
    }
    return 0;
}
```

Here argc is an integer which gives the number of arguments the user specified and argv is **pointer to a pointer**. We can understand this by remembering that the *char* type holds exactly one character. Thus a string of chars must use an array. Finally more than one argument can be given on the command line, hence argv is a pointer to the first argument, which is in turn a pointer to the first character of the first argument. Pointers can point to variables, functions and other pointers. *argv[i]* also the i-th argument. We can see this clearly by "playing" a little more:

```
#include <iostream>
#include <cstdlib> // provides atoi and atof.
using namespace std;

int main(int argc, char** argv) {
    // Example usage: test_main_args 8 10.234
    int i, j, arg_length;
    char *p;

    cout << "Number of arguments: " << argc << endl;

    for (i=1;i<argc;i++) {

        // Special behaviour of cout: if given a character pointer
        // it automatically dereferences it and prints until
        // the end of the character string.
        cout << "argument " << i << " = " << argv[i] << endl;

        p = argv[i]; // Set pointer equal to pointer of current argument.
        arg_length = strlen(p); // Gets length of character string.
        for (j=0;j!=arg_length;j++) {
            // Manually derefence p: prints just *p rather than *p to
            // end of string.
            cout << *p << endl;
            p++;
        }

    }
```

```
    return 0;
}
```

Dynamic arrays

Allocating a static array is fine when the array size is known at the time of compilation but frequently this is not a luxury available to us. Instead we need the flexibility to create arrays where the size is only known at run-time.

C++ provides the *new* and *delete* operators for handling dynamic memory allocation and deallocation. To allocate a variable-sized array we use the *new* keyword followed by the type of the array and its size in square brackets. An array allocated with *new* stays allocated until the program ends unless it is explicitly deleted. This is in contrast to other variables: the memory associated with static variables is released when that variable goes out of scope (i.e. at the block of code in which it is declared). It is good practice to delete arrays as needed, if only to maximise the available amount of memory! Deleting an array allocated using *new* is performed using *delete[]*. The square brackets tell delete to deallocate the entire array rather than just the first element of the array.

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

int main() {
    int n, *arr;
    cout << "Enter an integer: ";
    cin >> n;
    arr = new int[n];
    for (int i=0;i<n;i++) {
        arr[i] = 2*i+1;
        cout << arr[i] << endl;
    }
    delete[] arr;
    return 0;
}
```

A use for pointers

One use for pointers is with function arguments. By default C and C++ pass arguments by value (i.e. they copy the value of the arguments to new variables). This means that arguments are **local** to the function. Furthermore a function can only return a single variable:

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

int add(int i,int j) {
    // Return i+j
    return i+j;
}

int add_substract(int i,int j, int l) {
    // Place i-j in l, return i+j.
    l = i-j;
    cout << "l in function: " << l << endl;
    return i+j;
}
```

```
int main() {
    int i, j, k=0, l=0;
    i=42;
    j=24;
    k = add(i,j); // Fine.
    cout << "k from add: " << k << endl;
    k = add_substract(i,j,l); // Problem!
    cout << "k from add_subtract: " << k << endl;
    cout << "l from add_subtract: " << l << endl;
    return 0;
}
```

A solution is to use pointers when we want to allow variables to be modified within a function (this is also slightly more memory efficient).

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

int add_substract(int i,int j, int *l) {
    // Place i-j in l, return i+j.
    *l = i-j;
    cout << "*l in function: " << *l << endl;
    return i+j;
}

int main() {
    int i, j, k=0, l=0;
    i=42;
    j=24;
    k = add_substract(i,j,&l); // l in the main function now modified.
    cout << k << endl;
    cout << l << endl;
    return 0;
}
```

One important exception to how variables are passed: arrays are automatically passed by reference rather than value to save memory, hence:

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

int sum(int *coords, int ndim) { // Note *coords rather than coords!
    int total=0;
    for (int i=0;i<ndim;i++) {
        total+=coords[i];
    }
    return total;
}

int main() {
    int coords[3];
    for (int i=0;i<=2;i++) {
        coords[i] = 2*i+1;
        cout << "Component " << i << " of coords: " << coords[i] <<  endl;
```

```
    }
    cout << "Sum of coords: " << sum(coords,3) << endl; // No need to use &coords...
    return 0;
}
```

Task

8. Consider the following code:

```
#include <iostream>

using namespace std;

int main()
{
    int *a, *b, c, i;

    a = new int[3];
    for (i=0; i<3; i++) a[i] = i;

    b = a;
    c = (*b)++;

    return 0;
}
```

and

```
#include <iostream>

using namespace std;

int main()
{
    int *a, *b, c, i;

    a = new int[3];
    for (i=0; i<3; i++) a[i] = i;

    b = a;
    c = *b++;

    return 0;
}
```

What is the difference? Why?

9. Pointers represent one of the more conceptually challenging parts of C and C++. Familiarity with them is crucial for understanding much of the language (and existing code!), however. Experiment with the example programs and use the address and dereference operators in as many combinations as you can think of.

1 C does not have cin and cout for input and output. Instead, functions are used to achieve the same outcome and these are often seen in C++ code (as C++ contains all of C by construction). For example, the printf function (described here: http://www.cplusplus.com/reference/clibrary/cstdio/printf/) is used to for printing to STDOUT.