An oversimplified introduction to C++

23rd October 2014

Contents

Chapter 1

Why C++ - essential differences from C

1.1 Introduction

C++ is a programming language, just like FORTRAN, C, Pascal, BASIC, etc. It is more of a high level language than C, but retains all the 'low level' features of that language. So, its exact 'level' is somewhat suspect! The computer, of course, simply does not care. It changes internal switches according to 'OP Codes' (machine language) and has no idea *why* it is doing what it does. All high level languages are just for the convenience of us humans.

1.2 The need for object oriented programming (OOP)

The fastest programmes are written in assembly language. There was a time when that was a big difference. Computers are much faster nowadays and compilers are extremely complex. These wonderful programmes come with *optimisers* which generate code of such efficiency that it is almost impossible

to do better by writing programmes directly in assembly language[1]. As far as the different languages are concerned, as long as we use good compilers, the speeds will be effectively the same[2]. So for the purpose of performance, OOP languages like C++ are not really needed.

The real need is in the way the human brain works. 'Unstructured' programmes, like those written in assembly language, older varieties of FORTRAN (FORTRAN-4), etc. soon become full of so many jumps (*GO TO*) that after about 2000 lines of code, it becomes very difficult for humans to keep track, particularly if a project is shelved for some time and taken up later. 'Structured' code, like that using *C++*, *FORTRAN-77* (provided proper use is made of the *while* and *for* loops), etc., can be handled over a 25,000 line size, because logically linked regions are clearly demarcated in these languages. When code becomes much larger (maybe a million lines?) two fundamental barriers come into play. Firstly, our brains fail to keep the whole logic inside while developing the code - a level of *abstraction* and *encapsulation* is needed, with a clear hierarchy of constructs. Secondly, a team of programmers is now needed, each working on a different aspect of the problem. Clashes in naming and programming styles and approaches are

[1] Why should we learn assembly language then? The first reason is to be able to have complete control over the exact steps that the computer executes which is important in the case of timing loops, data acquisition, control and communication with other devices. The second reason - it makes us learn how the computer does its work and enables us to handle its internals! The third reason - to write compilers.

[2] However, *interpreters* are much, much slower. Languages like *Tcl/Tk*, *Python*, *BASIC* and many others, which commonly use interpreters, execute each command directly from the source code after compiling them (interpreting) line by line at the time of execution, and tend to be about a hundred times slower. *Perl* is an interpreter which keeps the compiled code ready for re-use during loops and discards it only on exit. It is as slow as any interpreted language during the first run, but is quite fast while executing repeated passes inside loops. *Java* uses an interesting concept called *byte-code*. The idea is to imagine a 'virtual machine' with a standardised set of *OP-codes* which are called *byte-codes*. Programmes written is *Java* are normally (OP-code compilers are also available) compiled into *byte-codes* for this virtual machine, which is the same across all platforms (different CPUs, different operating systems). During execution, there is a *Java virtual machine* which actually interprets these *byte-codes* into proper *OP-codes* on the host machine and runs them. Since the first level of compilation (into *byte-code*) has already simplified the programme substantially, the speed is not affected as badly as with pure interpreters. *Java byte-code* programmes run about twenty five times slower than genuinely compiled programmes. So this is a trade-off between speed and portability.

bound to occur. Object oriented programmes allow us to work under these situations. It is always better to develop programmes in an object oriented manner if we plan to use the code later or to let others use it.

1.3 Why C with a ++?

C++ is a superset of *C*, so any C programme is automatically a *C++* programme also. All the control structures (*if*, *while*, *for*, *switch*, etc.) remain the same, functions and subroutines work as usual and in this manual we will deal with the new features only. The first important difference in a *C++* programme is that whereas in the case of a proper *C* programme (fully POSIX compliant) all the variables have to be declared **before** the first line of executable code, in *C++* variables can be declared anywhere within the code. Memory allocation is dynamic as and when needed. Thus the following code is wrong in *C* but correct in *C++*:[3]

```
printf(“Give me the dimensions of the matrix:  “);
scanf(“%d%d”,&i,&j);
float halum[i][j];
```

Instead of using an extension of *.c* with the filename, we generally use *.C* or *.cpp* with *C++* programmes. For compilation the *g++* command is used instead of *gcc*[4].

1.4 Some features of *C++* different from *C*

1.4.1 Two currently used versions of *C++*

The older version of *C++* has been superseded by the newer one. In the old version, the headers had *.h* extensions. Now these are not used. Instead of

[3]The newer versions of the *gcc* compiler will allow declarations inside *C* programmes - but this is not strict *C* - do not take this behaviour for granted.

[4]*gcc* can also compile *C++* code, but many of the useful features will be absent because ‘*C++* binding’ (what is this?-AC) will not be used.

`#include <iostream.h>` we now have to use `#include <iostream>`. When we want to use our familier *C* headers, we may use the old ones, like `<stdio.h>`, `<math.h>`, `<string.h>`, `<stdlib.h>` or their newer versions: `<cstdio>`, `<cmath>`, `<cstring>`, `<cstdlib>`. If we use functions included in the `<cmath>` header, we no longer have to include the linker directive *-lm* in the newer version. For some of the headers, however,[5] there are no proper *C++* versions yet and we have to use the *.h* extension.

Another significant difference is the widespread use of *namespaces*. A namespace is a way of allowing different functions to have the same names. By placing these functions in different namespaces, ambiguity can be removed. Constructs like `cout` and `cin` have to be called with reference to their namespace, which is `std`. Thus *cout* becomes `std::cout` in the new version. In order to avoid this clumsy reference, we generally declare at the very outset that all the names that we will use will be with reference to the `std` namespace. This is done by including a line containing `using namespace std;` just after the headers.

1.4.2 Stream I/O

C is such a simple and rudimentary language that it does not have any built-in input/output capability! We have to use separate functions like `printf()` and `scanf()` for this purpose[6]. *C++* does have built-in support for I/O. The idea of a *stream* is as if there is a stream of data flowing along. You may *insert* data into the stream by using the *insertion operator* `<<` and *extract* data from the stream with the *extraction operator* `>>`. In order to understand this properly we have to wait till we discuss classes and objects, but for now we need only remember that with the inclusion of `<iostream>` we have automatic access to three streams, `cin`, `cout` and `cerr`.

[5] Mainly those dealing with deeper system calls - here *C* reigns supreme!

[6] These are functions because they have to be declared in header files and have to be called with parameters within brackets. Thus `pow(x,y)` is a function whereas `x*x` is not - it is a part of the language syntax.

1.4.2.1 Stream I/O on the console and keyboard

`cin`, `cout` and `cerr` correspond to the files `stdin`, `stdout` and `stderr` of *C*. The destination of the `cout` and `cerr` streams is the terminal and the source of the input stream `cin` is the keyboard[7]. The other very useful feature of these streams is that they are *intelligent*, that is, they can analyse the contents and take care of formatting automatically without any explicit instruction[8]. Below is a small example:

```
#include <iostream>

using namespace std;

int main(void)
{
float x = 4.3;
int i = 2;

cout << "See the fun :)  " << x << "\t" << i << endl;

return(0);
}
```

As you can see, we are using the insertion operator repeatedly to insert different kinds of entities into the `cout` stream, which ultimately shows them

[7]Unless they are redirected.
[8]There are ways of changing formats if we need to, see `iomanip`.

in readable form on the terminal[9].

For reading in data, `cin` is intelligent enough to convert the data on the input stream to the correct format depending on where (which kind of variable) the extraction operator is going to place the data. Thus:

```
cin >> x >> i;
```

would cause what you type on the keyboard to be converted to a `float` for `x` and an `int` for `i`.

1.4.2.2 Redirection of `cout`, `cerr` and `cin`

When you have a working programme with outputs on the console, you may want to divert the output to a file instead of seeing it on the console. This can be done easily from the command prompt. Thus:

```
$ ./bhutum > bhutum.out
```

will run `bhutum` and *redirect* its console output to `bhutum.out`. In a sililar manner:

```
$ ./bhutum < bhutum.in
```

will cause bhutum to take its input from the file `bhutum.in` instead of from the keyboard. If we want to keep the data that is already in a file and *append* to it instead of overwriting, we us the syntax:

[9] You could use "`\n`" instead of `endl` (no quotes required with `endl`) but there is a difference. Normally whatever goes into the stream is not shown immediately (The stream takes its own time to reach its destination, the work of the insertion operator is simply to throw something into the stream.) but the programme continues after the `cout` instruction which takes its own time to work. With `endl`, you are forcing a flush of the stream like `fflush(stdout)` in *C*. This may be useful for debugging - often a programme crashes after the `cout` instruction but you do not realise this because the `cout` has not caused a real write by the time the fatal error occurs. This is also the reason why `cerr` is configured to always flush itself immediately.

```
$ ./bhutum >> bhutum.out
```

and the new lines get added to the bottom of `bhutum.out` .

This rough and ready method has one major disadvantage. All output gets diverted - if there is a prompt for entering a particular value, it will also be sent to the output file and will not be shown. The way out is to use `cerr` for such output. `cerr` does not get diverted - only `cout` does. However, if we really want to divert `cerr`, use the syntax:

```
./bhutum 2> bhutum.err
```

and `cerr` will be redirected to bhutum.err. If you are really desperate:

```
./bhutum > bhutum.out 2> bhutum.err
```

will do the needful and oblige!

1.4.2.3 Stream I/O on files

For handling files you have to include `<fstream>` in the headers. Unlike `cout` and `cin`, which are already opened, you have to explicitly create an instance of type `ifstream` (for input) or `ofstream` (for output) with the name of the file as an argument. Once created, the objects can be used just like `cin` and `cout`:

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

using namespace std;

int main(void)
{
```

```
float x,y,z;
ifstream neoa(“halum.in”);
ofstream deoa(“halum.out”);

neoa >> x >> y;
cout << “x is “ << x << “ and y is “ << y << endl;
z = x*y+sin(x);
deoa << z << endl;

return(0);
}
```

What if I want to keep `neoa` ready but actually open the file later, say after receiving a name[10]? This is the equivalent of the separate instructions for creating a file pointer and using `fopen` to open an actual file in *C*:

```
ifstream neoa;
neoa.open(“halum.in”);
```

just now we do not need this.

[10]You will need this later when you do OOP.

Chapter 2

OOP and the concept of classes and objects

2.1 Object oriented programmes (OOP)

With *C++* three qualities characterise an object oriented programme:

1. Data encapsulation
2. Polymorphism
3. Inheritance

Instead of going into the rigorous definitions of these terms, let us try to understand the basic idea. Polymorphism means 'different processes same interface' or 'overloading'. Consider the addition of two numbers. If they are integers, their mode of storage is entirely different from that of real numbers (floats) and their algorithm for addition is different also. But when we use the '+' operator, do we care to remember this detail? Similarly, when we calculate a power of a number, the method for calculating an integral power and a fractional power is completely different. But we would like to use the same name and syntax for pow(2,3) and pow(2,3.3). In *C* if we gave the same name to two different functions, an error would occur. But since the arguments are of different types, in principle it should be possible to understand from the call itself which version is being invoked. The *C++* compiler

is smart enough to do this and will call the correct version as long as there is a way to tell them apart from their argument lists (types, number of arguments, etc). This is called overloading a function (actually overloading the name) or an operator and thus the requirement of polymorphism is satisfied. For the other two characteristics of OOP, we need the idea of classes.

2.1.1 Classes and objects

Let us recall the use of structures in *C*. At the first invocation a `struct` initiates the declaration of a new data type which is often a composite of many different variables, often of different types. Later invocations create variables of that type. Let us study an example:

```
struct bhutum {
char name[30];
char gender;
int no_of_teeth_left;
float height;
};

int main(void)
{
struct bhutum bhombol;

strcpy(bhombol.name,"Arani Chakravarti");
bhombol.gender = 'm';
bhombol.no_of_teeth_left = 3;
bhombol.height = 1.2;

return(0);
}
```

In *C++* we can use this structure as it is, but we can also use a class:

```
class bhutum {
public:
char name[30];
char gender;
int no_of_teeth_left;
float height;
};

int main(void)
{
bhutum bhombol;

strcpy(bhombol.name,"Arani Chakravarti");
bhombol.gender = 'm';
bhombol.no_of_teeth_left = 3;
bhombol.height = 1.2;

return(0);
}
```

What are the differences? In the first case, the trouble of declaring that `bhombol` is of type `bhutum` by appending a `struct` in front in the main programme is eliminated[1]. We need not put a `class` in front of `bhutum`; once the compiler finds that there is no pre-defined data type called `bhutum`, it will automatically search among the classes. The declaration `bhutum bhombol` is technically called an *instantiation* of `bhutum`, or creating an *instance* of `bhutum` and `bhombol` will be called an *object* of type `bhutum`.

The other difference lies in the `public:` directive. It says that the members of the class declared below it can be accessed from *outside the class*. Since the main programme is outside the class, this is what allows it to modify the values inside bhombol. In structures, the members are public

[1]We have learnt how to circumvent this limitation in *C* by using a `typedef`.

by default. In a class, the members are private by befault, although it is good practice to use the `private:` directive explicitly. If we used private variables instead of public ones in the programme above, the compiler would immediately signal an error. So a private variable is very safe inside a class, no one can do anything to it! But does this not make this variable useless?

This is where the other great feature of classes comes in - a class is an intelligent structure. A class can use its own variables and change them if it likes! How does in do so? By using subroutines which become members of the class just like the data and which can access all (including private) variables in the class. Now if these subroutines are placed under the `public:` directive, they can be called from the main programme and through them the main programme may be granted access to the private variables. What is the use of this round-about strategy? Let us see. Consider the following:

```
class bhutum {
private:
char name[30];
char gender;
int no_of_teeth_left;
float height;

public:
void set_name(char *cc)
{ strcpy(name,cc); }

char* get_name()
{ return(name); }

void set_gender(char c)
{ gender = c; }

char get_gender()
{ return(gender); }
```

```
void set_teeth_num(int i)
{ no_of_teeth_left = i; }

int get_teeth_num()
{ return(no_of_teeth_left; }

void set_height(float h)
{ height = h; }

float get_height()
{ return(height); }
};

int main(void)
{
bhutum bhombol;

bhombol.set_name(“Arani Chakravarti”);
bhombol.set_gender(’B’);
bhombol.set_teeth_num(-5);
bhombol.set_height(-200.54);

cout << “The gender of “ << bhombol.get_name() << “ is “ << bhom-
bol.get_gender() << “.”  << endl;

return(0);
}
```

Now this programme will compile properly and on running will produce the output:

```
The gender of Arani Chakravarti is B.
```

Note how the call to the member functions is just like accessing a member of the `bhombol` object. Due to the presence of the brackets, the compiler understands that this is really a subroutine call. Since the functions are of type `public`, they can be called from the main programme and they, in turn, are able to access the private members of the object. So far this is just doing simple things in a complex manner but notice that the gender was set to *B* which is meaningless. The values set for the number of teeth left and the height are also absurd. But now let us rewrite the `set_gender` method:

```
void set_gender(char g)
{
if(g == 'm')
{
gender = g;
}
else
{
gender = 'f';
}
}
```

So the default gender now is *f*. Now we see the essence of data protection. Since the `gender` variable is accessed only through a programme, we can put in as many checks and conditions on it as we want; so as far as we are concerned, the `gender` member has been imbued with intelligence! In the same way we can stipulate that a person cannot have a negative number of teeth and we are not really expected to be 100 feet tall :) . All this and some more characteristics[2] constitute *data encapsulation*. *C++* programmers generally start with a class and gradually keep adding capabilities to that class as and when the need arises. If there is need to add another member with the same name (same broad functionality) but with different kind of arguments,

[2]wait till we reach *inheritance*!

overloading (*polymorphism*) is used as in this example:

```
#include <iostream>

using namespace std;

class chhotoder_onko
{
private:


public:

void div(float a,float b)
{
if(b == 0)
{
cout << "Shunno diye bhaag hoinaa!"  << endl;
}
else
{
cout << "The result is " << a/b << "."  << endl;
}

void div(int a,int b)
{
if(b == 0)
{
cout << "Shunno diye bhaag hoinaa!"  << endl;
}
else
{
int quotient = a/b;
```

```
int remainder = a%b;
cout << ‘‘The quotient is ‘‘ << quotient << ‘‘ and the remainder is
‘‘ << remainder << ‘‘.’’  << endl;
}

void div(float a,int b)
{
if(b == 0)
{
cout << ‘‘Shunno diye bhaag hoinaa!’’  << endl;
}
else
{
cout << ‘‘The result is ‘‘ << a/b << ‘‘.’’  << endl;
}
};

int main(void)
{
chhotoder_onko sum;

sum.div(1,2);
sum.div(1.0,2.0);
sum.div(1.0,2);

return(0);
}
```

In each case, the correct version of `div` will be called. This opens up immense possibilities and prevents unintentional wrong use of code by others in a team who may not be very familiar with the internal structure of the subroutines.

2.1.2 Constructors and destructors

Initialisation of variables is a very important part of most programmes. Starting values for different variables often have to be put in place, files opened for subsequent operations and numbers which are potential divisors have to be set to non-zero values. Instead of doing these things from the main programme, a class can be set up to initialise its variables automatically. For this a public method is used, called the *constructor*. A constructor is declared with a name which is the same as that of the class. It does not have a return type but may have arguments. As soon as an instance of the class is created (by declaring that some object is of that type), the constructor runs once by itself[3]. Here is an example, where we also illustrate the utility of declaring a file stream before opening an actual file:

```
#include <iostream>
#include <fstream>

using namespace std;

class pook
{
private:

float a[100],b[100];
ifstream pik;
int i;

public:

pook(char *fname) // constructor
{
```

[3]You may think that as the name of the constructor subroutine is the same as that of the class, the declaration is also a call to the construtor.

```
i = 0;
pik.open(fname);
while(pik >> a[i] >> b[i] != NULL)
{
++i;
}
}

void show()
{
for(int j=0;j<i;++j)
{
cout << a[j] << "\t" << b[j] << "\n";
}
}
}; // end of class

int main(void)
{
pook pukai((char*)"in.dat");
pukai.show();
return(0);
}
```

Here we ask the constructor to open a file (in.dat) and keep all the data neatly organised in arrays just by creating an object named `pukai`. `i` is also initialised to 1 and then the number of lines can be found by looking at it[4].

Destructors are named with a tilde ('~') in front of the class name and do not take arguments (CHECK). They run when the programme ends or the object is destroyed. Thus, in the above case we could have written:

[4]If you want to acess `i` from the main programme then, of course, you need to have a public process for returning `i` in the class itself. `i` is a private variable.

```
~pook()
{
pik.close();
}
```

However this explicit closing is not generally needed in *C++*. In the case of interfacing with machines, it is quite likely that a whole set of closing down commands will have to be executed. Destructors come in handy in such cases.

2.1.3 Operators

Let us construct a three-dimensional vector class. We will include two constructors, one for creation of an object of type vector with values preloaded and the other for just creating the object without explicitly initialising its contents. We will also include a method for vector addition and one for printing out the values nicely:

```
#include <iostream>

using namespace std;

class vector
{
private:

float x,y,z;

public:

vector() // constructor for creating empty object
{ ; } // basically do nothing
```

```
vector(float a,float b,float c) // overloaded constructor for ini-
tialisation
{
x = a; y = b; z = c;
}

void show() // for a pleasing printout
{
cout << x << "i";
if(y < 0)
{ cout << y << "j"; }
else
{ cout << "+" << y << "j"; }
if(z < 0)
{ cout << z << "k"; }
else
{ cout << "+" << z << "k"; }
}

vector add(vector v)
{
vector temp;
temp.x = x+v.x;
temp.y = y+v.y;
temp.z = z+v.z;
return(temp);
}
};

int main(void)
{
float a,b,c;
```

```
cout << "Give me the first vector:  ";
cin >> a >> b >> c;
vector A(a,b,c); // uses the second constructor
cout << "Give me the second vector:  ";
cin >> a >> b >> c; // reuse the variables because the previous
// values have already been stored
vector B(a,b,c);
vector C; // uses the first constuctor - the memory is there but
not initialised

C = A.add(B); // vector addition

C.show();
return(0);
}
```

This is what happens - the add procedure of A is called with an argument of B and after the addition is performed a temporary vector is returned which is copied into C. Thus, we could have written B.add(A) also[5].

Now consider the syntax `A.add(B)` . What if we had named the function '+' instead of '`add`'[6]? The syntax would not be `A.+(B)` . If we could remove the dot and the brackets, the syntax would be our familiar `A+B`. This is what happens if you use the `operator` keyword:

```
vector operator +(vector v)
{
vector temp;
temp.x = x+v.x;
temp.y = y+v.y;
temp.z = z+v.z;
```

[5]But not in the case of subtraction!

[6]If you really try this the compiler will generate an error. The `operator` keyword is necessary.

```
return(temp);
}
```

If you now simply write C = A+B in the main programme, vector addition would be accomplished using our familiar syntax! For cross product and subtraction, we could write very similar procedures with the * and - operators. In the case of dot product, we need to use another operator symbol. We cannot use anything we like, the list of allowed operators is fixed. Let us use ^:

```
float operator ^(vector v)
{
return(x*v.x+y*v.y+z*v.z);
}
```

Only remember that we had to declare the return type as `float` because that is what the dot product is - we have to equate it to a number and will get an error if we try to equate it to a vector.

In a similar manner, you can create a complex class, matrix class (easy with fixed dimension, difficult with variable dimensions, but possible), classes tied to group multiplication tables and anything that you can imagine.

2.1.4 Inheritance

What if I want to modify a class? I may want to add new functionality or modify the way some already existant method works. The source code of the original class may not be available. Even if it is available, if the class is big, I will have to be very careful. I have to take care when I modify any private member so that the change does not compromise the working of some other method. I should not try to give an already existant name to a new variable - this will result in an error. If I have written my programmes separately, as soon as I incorporate the members in the old class, a huge list of errors will appear due to repeated names. If I change the names in the declarations,

I will have to be extra careful to get it changed in every instance of its use in my programmes, because the compiler will not point this out to me any more, there being already a variable with the old name in the original class. If some of the methods in the old class work satisfactorily for me, I may not want to spend time worrying whether my changes will affect them also. With really big programmes, these problems become almost insurmountable. The way out is through a beautiful feature of OOPs called inheritance.

I can declare that a new class is a descendant of the old class and inherits its features. The declaration is very simple:

```
class new_class :  public old_class
{
private:
.......
public:
.......
};
```

Now all the public methods of the original class automatically become public members of the new class and can be called just as if they are already built into the new class. However, the private members of the old class are not visible to the new class and so, if the new class has private members of the same name, they are actually completly different variables. The old variables are used by the functions that are inherited from the old class. I can add my variables and methods as needed and the new derived class will simply be the richer for it! If I want to change the way one of the old methods works, I simply have to include a new method with the same name in the new class. It will get precedence when called. If somewhere or the other I explicitly want a call to the version of a method in the old class (which has a different version with the same name in the new class) I will have to add a *scope resolution* directive: `old_class::method()` . Another class may be derived from this new class. It will inherit features right down the line. This is commonly used when a team of people work together : someone writes a *base class* and

someone else uses it with a *derived class*. In this way a whole hierarchy of inheritance can be built up with gradual increase in the capabilities of the later classes. It is also possible for a class to inherit directly from more than one old class:

```
class khagen :  public bhutum :  public very_old :  public not_so_old
```

However, multiple inheritance at the same level is discouraged by some authors (WHY?).

Another keyword that is sometimes used in this context is `protected`. A protected member acts like a public member, but if another class is derived from this class, it becomes a private member in the new class (CHECK).

Chapter 3

Miscellaneous topics

3.1 Passing variables by reference

How are variables passed from the main programme to a subroutine or from a calling subroutine to a called one? Basically through a stack. The calling programme places the variables one after another in a stack and then calls the subroutine. The subroutine 'pop's these variables off the stack, i.e. makes its own copy and works on them. Thus there is much copying of values going on during calls. This is called *passing by value*. There are other ways of sharing values. Inside a class, all the processes can access the private or public members. *Global variables*, which are declared outside the main programme or subroutines, can be accessed by any process below the declaration point. The name, of course, gets reserved and cannot be used for any other variable by code appearing subsequently.

In order to reduce the overhead (often very large for large objects) associated with back-and-forth copying *C++* allows *passing variables by reference*. Closely related to *pointers*, this is essentially a way of handling variables directly through their addresses. A reference variable is declared with an ampersand (&) appended to its name, and because it is a reference to some other variable, unlike pointers it has to be initialised at the time of declaration with the actual variable:

```
float x;
float &xr = x;
```

This is a *nonparameter declaration*. Once this is done, `x` and `xr` can be used as the same variable to all intents and purposes. However, this does not lead to any benefit.

The real saving of resources occurs when references are used for passing parameters. Consider the following programme:

```
#include <iostream>
using namespace std;
int main(void)
{
float x;
void bhutum(float&);
cout << "Give me x:  ";
cin >> x;
cout << "Before calling bhutum x is:  " << x << endl;
bhutum(x);
cout << "After calling bhutum x is:  " << x << endl;
return(0);
}
void bhutum(float &y)
{
y += 2.3;
cout << "This is bhutum, y = x+2.3, is:  " << y << endl;
}
```

In this programme, `bhutum` is declared and defined to take a reference variable. Even though it is called from the main programme with the variable `x`, actually the reference is passed. Thus if `x` was a large class, it would not be copied and would result in considerable improvement in performance. The programme also demonstrates another important and often risky feature.

Since the actual variable is referenced, and not a local copy, the subroutine changes the value of the calling parameter (which it calls `y` for its own use). If this is desired, well and good, but if it is done inadvertently, actual variable values will be changed and serious errors would occur. If the subroutine does not change the value of the calling variable, there will be no problem. One way to make sure that this does not happen is to use the `const` qualifier. If we declare:

```
void bhutum(const float &);
```

and define:

```
void bhutum(const float &y)
{
...
}
```

the above programme will throw up a compiler error because we are trying to change the constant parameter[1] `y`. Thus a better alternative to the way the vector operator + is defined in our vector class would be:

```
vector operator +(const vector &v).
```

3.2 Using *openmp* for parallelization

Computer clock speeds have all but stopped increasing. The hardware has become so complex that a very large amount of work is done using only a few clock cycles. There is not much left to optimize on that score. By more powerful computers, we basically mean more cores or more CPUs on the same motherboard. This leads to the idea of parallelization.

[1] Do not declare `x` as `const` in the main programme; you will not be able to change its value!

3.2.1 Not all programmes can be parallelized

The idea of running a programme in pieces over a large number of computers is particularly attractive. Thus, for instance, when two vectors are added together, the different components do not depend on each other and therefore, it is permissible to distribute the task among many computers, with each doing a part of the work and the final result being collected back by a 'master node'. But in the case of solving differential equations numerically, the output of the previous step becomes the input of the next, and the process is inherently serial. How to parallelize problems is a most important ongoing topic of research. Luckily, in physics a large variety of problems allow significant amounts of parallelism, e.g., tensor handling, Monte-Carlo simulations, genetic and other search algorithms etc. All the supercomputers in use today are basically large arrays of computers running in parallel.

3.2.2 MPI versus OpenMP

The most general mode for parallelism is by passing *messages* between different processors. A master node distributes arguments (through messages) to different cooperating nodes running their own copies of the same (sometimes different) programmes. The nodes do their work and send the results back to the master through messages. Here each node is in effect a different computer and the messages are passed using different protocols, often over the network. *MPI* (Message Passing Interface) is a famous example of this kind of parallelism. *PVM* (Parallel Virtual Machine) is an older system. However, for message passing to work, the code has to be written in a completely different way and network transmission takes time and hence the overhead is substantial. Parallelism at a very low level of granularity[2] cannot cause any improvement with these systems and often results in great degradation

[2]In principle, an expression like `y = sin(x)+ln(x)` can be parallelised with one machine calculating `sin(x)` and the other `ln(x)` at the same time. However, modern processors are so fast with mathematics that the time required to transmit `x` to the two different nodes and then getting the results back may be hundreds of times greater than the time taken by a node to calculate the individual terms. So jobs have to broken up into large 'chunks' for benefit, each of which takes significant CPU time.

of performance.

When many CPUs or cores sit on the same motherboard, they often share memory. Thus, one processor only needs to write something into a memory location for the other processors to be able to access it, eliminating the lengthy procedure of message passing. With such systems, it makes sense to parallelise portions within the same programme using the concept of *hyperthreading*. OpenMP is a protocol and library for doing this and modern *C* and *C++* compilers include support for OpenMP as a default[3].

3.2.3 Using OpenMP with *gcc* and *g++*

We have to use the `#pragma` directive. A `pragma` is a peculiar directive which instructs a compiler (at times the linker) to handle a portion of code in a particular manner. If the compiler does not support that `pragma` directive, it simply ignores it and so the code still runs properly. Let us inspect the following example:

```
#pragma omp parallel
cout << "halum\n";
```

For compiling this, we have to append `-fopenmp` at the end of the compilation command: `g++ -O -Wall bhoot.C -o bhoot.x -fopenmp` . If we run this snippet of code, halum is printed multiple times, since each core is handed this command! So this can be a way of finding out the nuber of cores. A better way would be to initialise an integer to zero and have the code line just increment it. Each core would get hold of the same integer (same memory location) and incerment it. Since the `pragma omp` directive works on only one line (CHECK), a `cout` following this line would print out the number of cores.

But this is terrible! We do not want our variables to have values according

[3]For very large programmes, the best approach is to use hyperthreading over CPUs sharing the same memory and then use message passing to connect many such clusters into a super cluster. Such programmes are called *hybrid programmes* and a considerable part of the current activity related to supercomputing is going on in this field.

to the number of cores available and the number of loops should not change also. For this we can use the following syntax:

```
#pragma omp parallel for
for(i=0;i<=420;++i)
{
something here
}
```

In this case, the `for` loop will be broken up into segments of `i` ranges and the cores asked to handle different segments in parallel. Due to the presence of the braces, the whole body of the loop will be considered one line and hence the pragma will work on it. The process can be demonstrated spectacularly if the loop body just prints the value of `i`. Each value of `i` will be printed only once, but if there are more than one core present, they will do their work in parallel, with different speeds depending on their individual loads (CHECK - do the cores multi-task in the middle of a `for` loop?) and as a result the numbers will not be printed in sequence. The speed will be increased considerably. Note that *this procedure can be used only if the sequence does not matter, i.e. the individual passes withing the loop are completely independent of each other.* Such a `for` loop (say some kind of initialisation or addition of components of matrices) is parallelizable and will benefit significantly from this simple `pragma` directive. In the wrong place this will wreak havoc. There are other directives to allow greater control over the parallelization; we do not discuss them here.

Index