Incompatibilities / Differences between C and C++

1. Objectives

1. The resolution operator
2. The operators new and delete
3. Inline functions
4. Functions returning a reference data type
5. Call by reference and call by value
6. The Boolean data type
7. Default parameters for functions
8. The reference type
9. Overloading functions

2. Incompatibilities between C and C++

2.1 Declaring and defining functions in C++

Alternative 1	**Alternative 2**

`#include "stdafx.h"`
`using namespace std;`
 	
`void twotimes (int a)`
`{`
`cout<<a;`
`a=a*2;`
`cout<<" doubled is "<<a;`
`}`
 	
`void main()`
`{`
`//system("color fc");`
`twotimes(10);`
`_getch();`
`}`
	`#include "stdafx.h"`
	`using namespace std;`
	
	`void twotimes(int);`
	
	`void main()`
	`{`
	`twotimes(10);`
	`_getch();`
	`}`
	
	`void twotimes(int a)`
	`{`
	`cout<<a;`
	`a=a*2;`
	`cout<<" doubled is "<<a;`
	`}`

Both alternatives are correct in C++.
Do not forget to declare the prototype of a function before call it.

2.2 Void pointers

There is NO default conversion of pointers from the type "void*" in other types of pointers.

For the type (void*) C++ allows the default conversion from:
the type of the pointer -> void*

Example:

```
void * vp;
int *ip;
vp = ip; //correct in C and in C++
ip=vp; //correct in C, incorrect in C++
ip = (int*)vp; //correct in C and C++
```

3. Improvements of C++

Suggestions: 1) Use more explicit names for functions and for variables.
Example:

```
int countSpaces(char* str)
{
int counter;
// some computations…
return counter;
}
```

Is more explicit and used than:

```
int cnt(char* str)
{
int i;
return i;
}
```

2) Try to add more comments when writing source code!
3) Do not put more than 90 characters on one line!

3.1 The boolean type

The boolean type is a basic data type in C++, C does not have this data type.
Example: a program which returns true if an element is found in an array and false otherwise.

```
bool find_elem()
{
bool found=false;
int i=0,n,elem, v[100];
cout<<"Introduce the length of the array "<<endl;
cin>>n;
while (i<n)
{
	cout<<"v["<<i<<"]= ";
	cin>>v[i];
	i++;
}
cout<<"Introduce the elementul you search for ";
cin>>elem;
for (i=0;i<n;i++)
{
if (elem==v[i])
 {
    found=true;
 }
}
return found;
}
```

3.2 I/O console (cin, cout)

- We use *cout* for printing on the screen

(cout – console output – stdout)

- We use *cin* for storing a new entry

(cin – console input – stdin)

Both are included in the *iostream* library. In order to use them we have to include the *iostream* library and we have to declare the use of the standard namespace std *using namespace std.*

3.3 Facilities to statements

In C++ you can declare the local variables anywhere inside the body of a function, but be carefull to declare them before using them!

Example:

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

int sum (int a, int b)
{
return (a+b);
}

void main()
{
   int x,y,z;
   cout<<"Introduce x and y "<<endl;
   cin>>x>>y;       //read  x and y
   cout<<"the sum of "<<x <<" and "<<y<<" is "<<sum(x,y);
                   //call the function directly in cout
     //only in the case when the function returns a value
     // (is not void)
      z=23+sum(10,23);
      cout<<"z= "<<z;
   char name[20];
      cout<<"What is your name? "<<endl;
      cin>>nume;
      cout<<"Wow! "<<nume<<" is such a beautiful name!";

}
```

3.4 Reference variables

In C++ we can declare identifiers as references to objects of a certain data type. This reference variable has to be instantiated when declaring it with the address of a variable which is already defined.

```
void main()
{
int number=10;
int &refint=number; //reference to int, HAS TO be instantiated
cout<<"the number is "<<number<<" and refint = "<<refint<<endl;
refint=200; //automatically number=200;
cout<<"AFTER.. the number = "<<numar<<" and refint = "<<refint<<endl;
}
```

An example of interchanging two numbers

The C version	The C++ version

`void intersch(int *a, int *b)`
`{`
`int aux;`
`aux=*a;`
`*a=*b;`
`*b=aux;`
`}`
 	
`void main()`
`{`
`int a,b;`
`cout<<"Give a and b "<<endl;`
`cin>>a>>b;`
`intersch(&a,&b);`
`cout<<"the interchanged values are "<<a <<" and "<<b;`
`}`
	`void intersch1(int &a, int&b);`
	
	`void main()`
	`{`
	`int a,b;`
	`      cout<<"Give a and b "<<endl;`
	`      cin>>a>>b;`
	`    intersch1(a,b);`
	`    cout<<"the interchanged values are "<<a <<" and "<<b;`
	`}`
	
	`void intersch1(int &a, int&b){`
	`int auxiliary;`
	`auxiliary = a;`
	`a = b;`
	`b = auxiliary;`
	`}`

If, for example, we have:

```
      int variabile;
      float &refvar=variabile;
```

they have different types. The compiler will not create a reference to a variable int, but will allocate memory for a float variabile, therefore a hidden object will be created for the given reference.

ATTENTION!! The data type of the reference variable and of the variable with which is instantiated has to be the same.

3 ways of parameter passing:

```
void function_call_by_value(int x, int y, int z)
{
x=10;
y=20;
z=30;
}
void function_call_by_pointer_reference(int *x, int *y, int *z)
{
*x=10;
*y=20;
*z=30;
}
void function_call_by_reference(int &x, int &y, int &z)
{
x=10;
y=20;
z=30;
}
void main()
{
int x=10, y=20, z=30;
int &refx=x;
int &refy=y;
int &refz=z;
function_call_by_value(x,y,z);
cout<<"By value  "<<x<<" "<<y<<" "<<z<<endl;

function_call_by_pointer_reference(&x,&y,&z);
cout<<"By pointer "<<x<<" "<<y<<" "<<z<<endl;

function_call_by_reference(refx,refy,refz);
cout<<"By reference "<<x<<" "<<y<<" "<<z<<endl;
}
```

3.5 Functions which return references

```
struct book
{
  char author[64];
  char title[64];
  float price;
};

book library[3] = {
  {"Jamsa and Klander", "All about C and C++", 49.9},
  {"Klander", "Hacker Proof", 54.9},
  {"Jamsa and Klander", "1001 Visual Basic Programmer's Tips", 54.9}};

book& give_abook(int i)
 {
   if ((i >= 0) && (i < 3))
     return(library[i]);
   else
     return(library[0]);
 }

void main(void)
 {
   cout << "Almost getting the book \n";
   book& a_book = give_abook(2);
   cout << a_book.author << ' ' << a_book.title;
   cout << ' ' << a_book.price;
}
```

3.6 Parameters with default values

In C++ we can declare functions with parameters which have default values. This allows us to call the function in several ways.

The arguments with default values have to be at the end of the parameter list. Example:

```
int divide(int a, int b=4)
{
int result;
result=a/b;
return result;
}

void function1(int, float=23.9, long=100);

int main()
{
      cout<< divide(15)<<endl;
      cout<< divide(20,10)<<endl;
      // cout<< divide(); //incorrect
      float f=4.5;
      function1(11); // correct call
      function1(11,f);// correct call
      function1(11,2.6); //correct call
      // functie(11,100); // incorrect call
   return 0;
}

void function1(int i, float f, long l)
{
cout<<"i= "<<i<<" f= "<<f<<" l= "<<l<<endl;
}
```

3.7 Overloading function names

C++ allows us to have two or more functions with the same name, but with a different number of parameters and a different return value.
Example:

```
#include "stdafx.h"
using namespace std;
```

```
int operation(int a, int b)
{
return (a*b);
}

float operation(float a, float b)
{
return (a/b);
}

char * operation(char *s1, char *s2)
{
char *result = new char[strlen(s1)+strlen(s2)+1];
strcpy(result, s1);
strcat(result, s2);
return result;
}

int main ()
{
int x=5,y=2;
float n=5.0,m=2.0;
cout << operation(x,y);
cout << "\n";
cout << operation(n,m);
cout << "\n";
char *s1="The first string ", *s2="The second string ";
cout<<"Adding to strings "<<operation(s1,s2)<<endl;
return 0;
}
```

3.8 Inline functions

```
Syntax:     inline data type name (parameters ... )
                                {
                                  instructions ...
                                }
```

You can use the inline functions when you have a small number of parameters and not many instructions. The advantage of using inline functions is an increased execution speed.

The inline functions replace the macros built using #define and avoid the difficulties caused by these macros:
- these functions do not return values
- the parameters should be specified in parentheses, for an assessment with respect to the precedence of the operators.

Ex:
#define Max(a,b) ((a)<(b)) ? (b) : (a)

```
inline int Max(int a, int b)
{
return (a<b) ? b : a ;
}
```

Study the following example which prints the running time when calling 30000 times each function

```
#include "stdafx.h"
#include <time.h>
using namespace std;

inline void swap_inline(int *a, int *b, int *c, int *d)
 {
   int temp;

   temp = *a;
   *a = *b;
   *b = temp;

   temp = *c;
   *c = *d;
   *d = temp;
 }

void swap_call(int *a, int *b, int *c, int *d)
 {
   int temp;

   temp = *a;
   *a = *b;
   *b = temp;

   temp = *c;
   *c = *d;
   *d = temp;
 }


void main(void)
 {
   clock_t start, stop;
   long int i;
   int a = 1, b = 2, c = 3, d = 4;

   start = clock();
   for (i = 0; i < 300000L; i++)
     swap_inline(&a, &b, &c, &d);
   stop = clock();
   cout << "Time for inline: " << stop - start;

   start = clock();
   for (i = 0; i < 300000L; i++)
     swap_call(&a, &b, &c, &d);
   stop = clock();
   cout << "\nThe running time for calling the function is: " << stop -
start;
 }
```

3.9 New operators: new and delete

In C the memory management is implemented in auxiliary libraries and the user can use some functions. Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But, what if we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space. In C++ we have two new operators for memory management (dynamic memory):
- new – for memory allocation
- delete – the memory is freed

Ex:

```
int main()
{
char * str = new char [100];
delete [] str;
}
```

What is the effect of the following program?

```
void main(void)
 {
   char *array = new char[256];
   int i;

   for (i = 0; i < 256; i++)
     array[i] = 'A';

   for (i = 0; i < 256; i++)
     cout << array[i] << ' ';
 }
```

3.10 The resolution operator

Example:

```
#include "stdafx.h"
using namespace std;

int global_variable=300;

void main()
{
int global_variable=100;
cout<<"the value of the local variable is "<< global_variable<<endl;
cout<<" the value of the global variable is "<<::global_variable<<endl;
}
```

Keywords in C++

asm
auto
bad_cast
bad_typeid
bool
break
case
catch
char
class
const
const_cast
continue
default
delete
do
double
dynamic_cast
else
enum
except
explicit
extern
false
finally
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
operator
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
type_info
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
while
xalloc

Remark: The keywords cannot be used as names for variabiles, functions, classes etc.

Operators in C++

The C++ operators (from the highest to the lowest priority) are:

::	Scope resolution	None
::	Global	None
[ ]	Array subscript	Left to right
( )	Function call	Left to right
( )	Conversion None	
.	Member selection (object)	Left to right
->	Member selection (pointer)	Left to right
++	Postfix increment	None
--	Postfix decrement	None
new	Allocate object	None
delete	Deallocate object	None
delete[ ]	Deallocate object	None
++	Prefix increment	None
--	Prefix decrement	None
*	Dereference	None
&	Address-of	None
+	Unary plus	None
-	Arithmetic negation (unary)	None
!	Logical NOT	None
~	Bitwise complement	None
sizeof	Size of object	None
sizeof ( )	Size of type	None
typeid( )	type name	None
(type)	Type cast (conversion)	Right to left
const_cast	Type cast (conversion)	None
dynamic_cast	Type cast (conversion)	None
reinterpret_cast	Type cast (conversion)	None
static_cast	Type cast (conversion)	None
.*	Apply pointer to class member (objects)	Left to right
->*	Dereference pointer to class member	Left to right
*	Multiplication	Left to right
/	Division	Left to right
%	Remainder (modulus)	Left to right
+	Addition	Left to right
-	Subtraction	Left to right
<<	Left shift	Left to right
>>	Right shift	Left to right
<	Less than	Left to right
>	Greater than	Left to right
<=	Less than or equal to	Left to right
>=	Greater than or equal to	Left to right
==	Equality	Left to right
!=	Inequality	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
e1?e2:e3	Conditional	Right to left
=	Assignment	Right to left
*=	Multiplication assignment	Right to left
/=	Division assignment	Right to left

%=	Modulus assignment	Right to left
+=	Addition assignment	Right to left
-=	Subtraction assignment	Right to left
<<=	Left-shift assignment	Right to left
>>=	Right-shift assignment	Right to left
&=	Bitwise AND assignment	Right to left
|=	Bitwise inclusive OR assignment	Right to left
^=	Bitwise exclusive OR assignment	Right to left
,	Comma	Left to right

Remarks:
a) Operators can be oveloaded.
b) except for the following ones: . , .* , :: , ? : , # , ##.
c) New operators introduced: new, delete, .*, ->,*.

4. Problems

4.1 Print on the screen a text from a txt file “test.txt”

```
int main()
{
//system("color fc");
char ch;
FILE* fp=fopen("test.txt", "r");
if(fp==NULL) {
cout<<"Error when trying to open the text file" <<endl;
return 0;
}
while((ch=fgetc(fp))!=EOF)
cout<<ch;
fclose(fp);

_getch();
return 0;
}
```

4.2 Search how many times a word (introduced by the user) occurs in a text file.

4.3 Write a program which decreasingly sorts a set of words. The user introduces the words. For each operation define a separate function.