Programming in C++

C++ is a superset of an improved C language, which was extended with OOP constructs. C was developed in the early 1970's, and applied to the development of the UNIX operating system.

Functions

As C is a functional language, it is based on the definition of functions. A function takes a number of arguments, and returns one value or none (void). C++ employs functions as well, but mainly as methods of classes.

Functions are declared by naming similarly as variables, preceded by return-value type, and followed by arguments enclosed in parentheses:

`int sum(int to);`	**Declaration of sum with one argument**
`int bar();`	**Declaration of bar with no arguments**
`void foo(int i, float x);`	**Declaration of foo with two arguments**

To define a function, a body of statements is enclosed in braces:

```
int sum( int to ) {
    int res=0;
    for ( int i=1; i<=to; i++ )
        res += i;
    return res;
}

int bar() {
    int a, b;
    cin >> a >> b;
    return (a+b)/2;
}
```

Program Structure

Source programs in C consist of compiler directives, global declarations, structs, and function definitions. C++ added class definitions.

Any number of functions can be defined in any order, of which one must be named "main" where execution starts and ends. Functions can be called from within other functions, providing they have already been declared. Library functions are declared by including header files.

Example: Calculation of permutations ${}_nP_r = \frac{n!}{(n-r)!}$

```
#include <iostream.h>
#include <stdlib.h>
int factorial( int n );

void main() {
    int r, n;
    cout << "Enter r and n: ";
    cin >> r >> n;
    float p = factorial(n) / factorial(n-r);
    cout << "Permutation" << r << " of"
         << n << " =" << p << '\n';
}
int factorial( int n ) {
    n = abs(n);
    int fac = 1;
    for ( int i=2; i<=n; i++ )
        fac *= i;
    return fac;
}
```

<u>Tutorial:</u> Extend the example above by the following:

1. **Write a function to return ${}_nP_r$ and use it in main above.**
2. **Write a function to return ${}_nC_r = \frac{n!}{r!\,(n-r)!}$.**
3. **Replace 1&2 by one function to return either P or C according to a flag variable, and use it in a modified main function.**

Argument Passing

Argument passing is the primary way of providing functions with the variable data to operate on. Function headers usually include a dummy argument list enclosed in parentheses:

```
<return-type> <function-name> ( <type> <var> , . . . )
```

When a function is called, the call statement must supply corresponding values (constants or pre-defined variables) of matching types.

```
<function-name> ( <value> , . . . )
```

C++ allows passing of arguments by value only. Each value in the call list is copied to define the corresponding dummy that becomes a local variable within the function scope. To pass by reference indirectly, pointers (addresses) are used.

Basic data types

Argument lists may include built-in basic types of variables, declared simply by preceding their names with the type keyword, e.g.:

```
void func (char c, int x, float a, float b);
```

A call to func must supply the four required values, e.g.:

```
func('a', y, float(z), 5.0+z);
```
//y&z are pre-defined integers

User-defined data types

A record data structure in C (struct) can be used to combine several variables of possibly different types together, e.g.:

```
struct time {
    int h, m;
    float s; };
```

Considering "struct time" as a new type, variables of this type can be declared, e.g.:

```
struct time work, sleep;
```

Variables of struct types are initialized by enclosing a list of member values in order within braces, or defined individually using the dot operator, e.g.:

```
sleep = { 8, 44, 22.6 };
sleep.h = 10;
sleep.m = sleep.m + 20;
```

Functions may use struct variable arguments, so that when called, a pre-defined struct variable is copied to define a corresponding dummy.

Example: Define a time struct of int hours and minutes. Write a function to obtain the difference between two given times in minutes. Add a main function to calculate difference between a given time and a fixed time at 10:30.

```
#include <iostream.h>

struct time {
    int h, m;
};

int timediff(struct time t1, struct time t2) {
    return (t1.h-t2.h)*60+(t1.m-t2.m);
}

void main() {
    struct time f={10,30}, g;
    cout << "Enter time in hr & min: ";
    cin >> g.h >> g.m;
    cout << "Time difference = "
         << timediff( g, f ) << " min\n";
}
```

Tutorial: Extend the example above by the following functions, and modify main to test them:

1. **Function for difference between two given times in hours.**
2. **Function to add time t2 to time t1.**
3. **Function for the resulting time from addition of two given times.**

Arrays and Pointers

Arrays are declared using the subscript operator [], e.g.:

```
float x[50];          x is an array of 50 real elements
int m[5][8];          m is a two-dimensional array of 5x8 integers
char name[30];        name is an array of 30 characters (bytes)
struct time t[5];     t is an array of 5 elements of struct time
```

Strings are arrays of char, terminated by an ASCII 0 or ('\0'). They are defined by enclosing characters in double quotes, e.g.:

```
char name[30] = "ahmed";  declares 30 elements and defines first 6
char name[] = "ahmed";    declares and defines 6 elements
```

Pointers are variables that contain addresses. They are declared by putting an asterisk after the data type they point to, e.g.:

```
char * str;           str is pointer to char
int *xp, *yp;         xp and yp are pointers to integers
```

The array name is a pointer to the first element of the array, e.g.:

```
int a[30];            a is a pointer defined as &a[0]
char s[] = "ahmed";   s is a pointer defined as &s[0]
```

Array elements are referenced by either subscripts or pointers, e.g.:

```
cout<<a[0];  or  cout<<*a;        prints first element of a
s[3]='a';    or  *(s+3)='a';      changes string s to "ahmad"
```

Functions may use pointer arguments, which are passed address values when called. This enables referencing variables of the caller function.

Tutorial: Write the following:

1. **Function to return the greater of two given integers a & b. Add main to set a>b always.**
2. **Function that sets a>b for any two integers a & b in the main.**
3. **Function to invert an integer array. Add main to test it.**
4. **Program to append character 's' to a given string.**

Appendix
C++ Grammar Tables

Data Types

The following table describes the basic data types of C++. The "Size" in bytes and "Domain" values are dependent on the system used, with specified values given below for a 386 PC running Linux. Therefore, those values should only be considered relatively.

Type	Description	Size	Domain
char	Signed character/byte. Characters are enclosed in **single** quotes.	1	-128..127
double	Double precision number	8	ca. $10^{-308}..10^{308}$
int	Signed integer	4	$-2^{31}..2^{31}-1$
float	Floating point number	4	ca. $10^{-38}..10^{38}$
long (int)	Signed long integer	4	$-2^{31}..2^{31}-1$
long long (int)	Signed very long integer	8	$-2^{63}..2^{63}-1$
short (int)	Short integer	2	$-2^{15}..2^{15}-1$
unsigned char	Unsigned character/byte	1	0..255
unsigned (int)	Unsigned integer	4	$0..2^{32}-1$
unsigned long (int)	Unsigned long integer	4	$0..2^{32}-1$
unsigned long long (int)	Unsigned very long integer	8	$0..2^{64}-1$
unsigned short (int)	Unsigned short integer	2	$0..2^{16}-1$

The size of a data type on any particular system can be obtained with the sizeof operator, e.g.: `cout << sizeof( long );`

Operators

The following table includes C++ operators, each with its priority or precedence and the order of evaluation.

Operator	Priority	Description	Order
()	1	Function call operator	from left
[]	1	Subscript operator	from left
− >	1	Element selector	from left
!	2	Boolean NOT	from right
~	2	Binary NOT	from right
++	2	Post-/Preincrement	from right
− −	2	Post-/Predecrement	from right
−	2	Unary minus	from right
(*type*)	2	Type cast	from right
*	2	Derefence operator	from right
&	2	Address operator	from right
sizeof	2	Size-of operator	from right
*	3	Multiplication operator	from left
/	3	Division operator	from left
%	3	Modulo operator	from left
+	4	Addition operator	from left
−	4	Subtraction operator	from left
<<	5	Left shift operator	from left
>>	5	Right shift operator	from left
<	6	Lower-than operator	from left
<=	6	Lower-or-equal operator	from left
>	6	Greater-than operator	from left
>=	6	Greater-or-equal operator	from left
==	7	Equal operator	from left
!=	7	Not-equal operator	from left
&	8	Binary AND	from left
^	9	Binary XOR	from left
|	10	Binary OR	from left
&&	11	Boolean AND	from left
||	12	Boolean OR	from left
?:	13	Conditional operator	from right
=	14	Assignment operator	from right
op=	14	Operator assignment operator	from right
,	15	Comma operator	from left

Control Statements

In addition to declarations, definitions, assignments, calls & messages, C++ defines all the usual flow control statements. Statements are terminated by a semicolon ";". Multiple statements can be grouped into blocks by enclosing them in braces "{" and "}". Thus, *stmt* below is either a singular statement that ends with a semicolon, or a multiple statement block ending in a closing brace.

Statement	Description
`break;`	Leave current block. Also used to leave `case` statement in `switch`.
`continue;`	Only used in loops to continue with next loop immediately.

`do`
stmt
`while (`*expr*`);`
	Execute *stmt* as long as *expr* is TRUE.

`for ([`*expr*`]; [`*expr*`]; [`*expr*`])` *stmt*	This is an abbreviation for a `while` loop where the first *expr* is the initialization, the second *expr* is the condition and the third *expr* is the step.
`goto` *label*;	Jumps to position indicated by *label*. The destination is *label* followed by colon ":".
`if (`*expr*`)` *stmt* [`else` *stmt*]	IF-THEN-ELSE in C notation
`return` [*expr*];	Return from function. If function returns `void` `return` should be used without additional argument. Otherwise the value of *expr* is returned.

`switch (`*expr*`) {`
`case` *const-expr*`:` *stmts*
`case` *const-expr*`:` *stmts*
...
[`default:` *stmts*]
}
	After evaluation of *expr* its value is compared with the `case` clauses. Execution continues at the one that matches. BEWARE: You **must** use `break` to leave the `switch` if you don't want execution of following `case` clauses! If no `case` clause matches and a `default` clause exists, the statements of the default clause are executed.

`while (`*expr*`)` *stmt*	Repeat *stmt* as long as *expr* is TRUE.