C++ : As a Better C

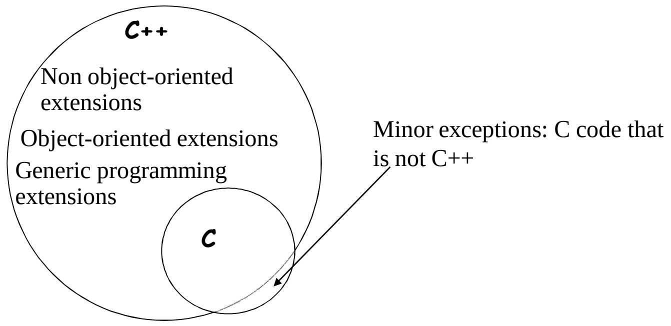
C++ was developed from the C programming language, by adding some features to it. These features can be collected in three groups:

1. Non-object-oriented features, which can be used in coding phase. These are not involved with the programming technique.

2. Features which support object-oriented programming.

3. Features which support generic programming.

With minor exceptions, C++ is a superset of C.


C++
Non object-oriented extensions
Object-oriented extensions
Generic programming extensions
C
Minor exceptions: C code that is not C++

C++'s Enhancements to C (Non Object-Oriented)

Caution: The better one knows C, the harder it seems to be to avoid writing C++ in C style, thereby losing some of the potential benefits of C++.

1. Always keep object-oriented and generic programming techniques in mind.

2. Always use C++ style coding technique which has many advantages over C style.

Non object-oriented features of a C++ compiler can be also used in writing procedural programs.

Single-Line Comments:

C++ allows you to begin a comment with **//** and use the remainder of the line for comment text.

c = a + b ;   **//** *This is a comment*

Declarations and Definitions in C++:

Remember; there is a difference between a *declaration* and a *definition* .

A ***declaration*** introduces a name – an identifier – to the compiler. It tells the compiler "This function or this variable exists somewhere, and here is what it should look like."

A ***definition***, on the other hand, says: "Make this variable here" or "Make this function here." It allocates storage for the name.

Example:

```
extern int i;                // Declaration
int i;                       // Definition

struct ComplexT{             // Declaration
    float re,im;
};
ComplexT c1,c2;              // Definition

void func( int, int);        // Declaration     (its body is a definition)
```

- In *C*, declarations and definitions must occur at the beginning of a block.
- In *C++* declarations and definitions can be placed anywhere an executable statement can appear, except that they must appear prior to the point at which they are first used. This improve the readability of the program.
- A variable lives only in the block, in which it was defined. This block is the **scope** of this variable.

```
int a=0;
for (int i=0; i < 100; i++){   // i is defined at the beginning of the for loop
      a++;
      int p=12;                // Definition of p
      ...                      // Scope of p
}                              // End of scope for i and p
```

Scope Operator (::)

A definition in a block (local name) can hide a definition in an enclosing block or a global name.

It is possible to use a hidden global name by using the scope resolution operator ::

```
int y=0;          // Global y
int x=1;          // Global x
void f(){         // Function is a new block
   int x=5;       // Local  x=5, it hides global x
   ::x++;         // Global x=2
   x++;           // Local x=6
   y++;           // Global y=1, scope operator is not necessary
}
```

Caution: It is not recommended to give identical names to global and local data, if it is not mandatory.

Like in *C*, in *C++* the same operator may have more than one meaning. The scope operator has also many different tasks, which are presented in following chapters.

Namespaces

When a program reaches a certain size it's typically broken up into pieces, each of which is built and maintained by a different person or group.

Since C effectively has a single arena where all the identifier and function names live, this means that all the developers must be careful not to accidentally use the same names in situations where they can conflict.

The same problem come out if a programmer try to use the same names as the names of library functions.

Standard C++ has a mechanism to prevent this collision: the **namespace** keyword.

Each set of C++ definitions in a library or program is "wrapped" in a namespace, and if some other definition has an identical name, but is in a different namespace, then there is no collision.

Example:

```
namespace programmer1{          // programmer1's namespace
    int iflag;                  // programmer1's iflag

    void g(int);                // programmer1's g function
    :                           // other variables
}                               // end of namespace

namespace programmer2{          // programmer2's namespace
    int iflag;                  // programmer2's iflag
    :
}                               // end of namespace
```

Accessing the variables:

```
programmer1::iflag = 3;         // programmer1's iflag
programmer2::iflag = -345;      // programmer2's iflag
programmer1::g(6);              // programmer1's g function
```

If a variable or function does not belong to any namespace, then it is defined in the global namespace. It can be accessed without a namespace name and scope operator.

using declaration:

This declaration makes it easier to access variables and functions, which are defined in a namespace.

```
using programmer1::iflag;       //  applies to a single item in the namespace
iflag = 3;                      //  programmer1::iflag=3;
programmer2::iflag = -345;
programmer1::g(6);
```

Or

```
using namespace programmer1;    // applies to all elements in the namespace
iflag = 3;                      // programmer1::iflag=3;
g(6);                           // programmer1's function g
programmer2::iflag =  -345;
```

Standard C++ header files

In the first versions of C++, mostly '**.h**' is used as extension for the header files.

As C++ evolved, different compiler vendors chose different extensions for file names (.hpp, .H , etc.). In addition, various operating systems have different restrictions on file names, in particular on name length.

These issues caused source code portability problems.

To solve these problems, the standard uses a format that allows file names longer than eight characters and eliminates the extension.

For example, instead of the old style of including **iostream.h**, which looks like this: #include <iostream.h>
you can now write: #include <iostream>

The libraries that have been inherited from C are still available with the traditional '**.h**' extension. However, you can also use them with the more modern C++ include style by puting a "**c**" before the name. Thus:

#include <stdio.h>	become:	#include <cstdio>
#include <stdlib.h>		#include <cstdlib>

In standard C++ headers all declarations and definitions take place in a namespace : **std**

If you use standard headers without extension you have to write:

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

Today most of C++ compilers support old libraries and header files too. So you can also use the old header files with the extension '.h'.

For a high-quality program prefer always the new libraries.

For your own header files you may still use the extension '.h'.

Example: #include "myheader.h"

INPUT/OUTPUT

Instead of library functions (printf, scanf), in C++ library objects are used for IO operations.

When a C++ program includes the **iostream** header, four objects are created and initialized:

cin handles input from the standard input, the keyboard.

cout handles output to the standard output, the screen.

cerr handles unbuffered output to the standard error device, the screen.

clog handles buffered error messages to the standard error device

Using **cout** object

To print a value to the screen, write the word **cout**, followed by the insertion operator (**<<**).

```
#include<iostream>                  // Header file for the cout object
int main() {
  int i=5;                          // integer i is defined, initial value is 5
  float f=4.6;                      // floating point number f is defined, 4.6
  std::cout << "Integer Number = " << i << " Real Number= " << f;
  return 0;
}
```

See Example e21.cpp

Using **cin** object

The predefined **cin** stream object is used to read data from the standard input device, usually the keyboard. The **cin** stream uses the **>>** operator, usually called the "get from" operator.

```
#include<iostream>
using namespace std;                // we don't need std:: anymore
int main() {
  int i, j;                         // Two integers are defined
  cout << "Give two numbers \n";    // Message to screen, cursor to the new line
  cin >> i >> j;                    // Read i and j from the keyboard
  cout << "Sum= " << i + j << "\n"; // The sum of the numbers to the screen
  return 0;
}
```

See Example e22.cpp

The Type bool

The type bool represents boolean (logical) values, for which the reserved values true and false may be used.

Before **bool** became part of Standard C++, everyone tended to use different techniques in order to produce Boolean-like behavior. These produced portability problems and could introduce subtle errors.

Because there's a lot of existing code that uses an **int** to represent a flag, the compiler will implicitly convert from an **int** to a **bool** (nonzero values will produce **true** while zero values produce **false**).

Do not prefer to use integers to produce logical values.

Example:

```
bool is_greater;           // Boolean variable: is_greater
is_greater = false;        // Assigning a logical value
int a,b;
.................
is_greater = a > b;        // Logical operation
if (is_greater) ......     // Conditional operation
```

Constants

In standard C preprocessors directive #define is used to create constants.

```
#define  PI        3.14
```

C++ introduces the concept of a named constant that is just like a variable, except that its value cannot be changed.

The modifier **const** tells the compiler that a name represents a constant.

```
const int MAX = 100;       // MAX is constant and its value is 100.
...
MAX = 5;                   // Compiler Error!  Because MAX is constant
```

const can take place before (left) and after (right) the type. They are always (both) allowed and equivalent.

```
int const MAX = 100;       // The same as      const int MAX = 100;
```

The keyword const very often occurs in ***C++*** programs as we will see in this course. This usage decreases error possibilities.

To make your programs more readable, use uppercase font for constant identifiers.

Another usage of the keyword const is seen in the declaration of pointers.

There are three different cases:

a) The data pointed by the pointer is constant, but the pointer itself however may be changed.

```
const char *p = "ABC";       // Constant data= "ABC", pointer is not const
```

p is a pointer variable, which points to chars. The const word may also be written after the type:

```
char const *p = "ABC";       // Constant data= "ABC", pointer is not const
```

Whatever is pointed to by p may not be changed: the chars are declared as const. The pointer p itself however may be changed.

```
*p = 'Z';           // Compiler Error! Because data is constant
p++;                // OK, because the address in the pointer may change.
```

b) The pointer itself is a const pointer which may not be changed. Whatever data is pointed to by the pointer may be changed.

```
char * const sp = "ABC";    // Pointer is constant, data may change
*sp = 'Z';                  // OK, data is not constant
sp++;                       // Compiler Error! Because pointer is constant
```

c) Neither the pointer nor what it points to may be changed

```
const char * const ssp = "ABC";   // Pointer and data are constant
*ssp = 'Z';                       // Compiler Error! Because data is constant
ssp++;                            // Compiler Error! Because pointer is const
```

The same pointer definition may also be written as follows:

char **const** * **const** ssp = "ABC";

The definition or declaration in which const is used should be read from the variable or function identifier back to the type identifier:

"ssp is a const pointer to const characters"

Casts

Traditionally, ***C*** offers the following *cast* construction:

- (typename) expression

  Example: f = (float) i / 2; *// i is int and f is float*

Following that, *C++* initially also supported the *function call style* cast notation:

- typename(expression)

Example: Converting an integer value to a floating point value

```
int    i=5;        // i is an integer. Initial value is 5.
float  f;          // f is an floating point variable.
f = float(i)/2;    // first, i is converted to float and then divided by 2.
```

But, these casts are now called *old-style casts*, and they are deprecated. Instead, four *new-style casts* were introduced:

a) static_cast:

The static_cast<type>(expression) operator is used to convert one type to an acceptable other type.

```
int    i=5;                        // i is an integer. Initial value is 5.
float  f;                          // f is an floating point variable.
f = static_cast<float>(i)/2;       // i is converted to float and divided by 2.
```

b) const_cast:

The const_cast<type>(expression) operator is used to do away with the const-ness of a (pointer) type.

In the following example p is a pointer to constant data, and q is a pointer to non-constant data. So the assignment q = p is not allowed.

```
const char *p = "ABC";    // p points to constant data
char        *q;           // data pointed by q may change
q = p;                    // Compiler Error! Constant data may change
```

If the programmer wants to do this assignment on purpose then he/she must use the const_cast operator:

```
q = const_cast<char *>(p);
*q = 'X';                          // Dangerous?
```

c) reinterpret_cast:

The reinterpret_cast<type>(expression) operator is used to reinterpret byte patterns.

For example, the individual bytes making up a structure can easily be reached using a reinterpret_cast<>().

```
struct S{                                   // A structure
  int i1, i2;                               // made of two integers
};

int main(){
  S    x;                                   // x is of type S
  x.i1=1;                                   // fields of x are filled
  x.i2=2;
  unsigned char *xp;                        // A pointer to unsigned chars
  xp = reinterpret_cast<unsigned char *> (&x);      // reinterpretation
  for (int j=0; j<8; j++)                   // bytes of x on the screen
    std::cout <<  static_cast<int>(*xp++);
  return 0;
}
```

The structure S is made of two integers (2x4=8 bytes). x is a variable of type S. Each byte of x can be reached by using the pointer xp.

d) dynamic_cast:

The dynamic_cast<>() operator is used in the context of inheritance and polymorphism. We will see these concepts later. The discussion of this cast is postponed until the section about polymorphism.

Comments:

• Using the cast-operators is a dangerous habit, as it suppresses the normal type-checking mechanism of the compiler.

• It is suggested to prevent casts if at all possible.

• If circumstances arise in which casts have to be used, document the reasons for their use well in your code, to make double sure that the cast is not the underlying cause for a program to misbehave.

Dynamic Memory Allocation

In ANSI C, dynamic memory allocation is normally performed with standard library functions malloc and free.

The C++ **new** and **delete** operators enable programs to perform dynamic memory allocation more easily.

The most basic example of the use of these operators is given below.
An int pointer variable is used to point to memory which is allocated by the operator new.
This memory is later released by the operator delete.

```
int *ip;                    // A pointer to integers
ip = new int;               // Memory allocation
    ......
delete ip;                  // Releasing the memory
```

Note that new and delete are operators and therefore do not require parentheses, as required for functions like malloc() and free().

• The operator new returns a pointer to the kind of memory that's asked for by its argument (e.g., a pointer to an int in the above example).

• Note that the operator new uses a type as its operand, which has the benefit that the correct amount of memory, given the type of the object to be allocated, becomes automatically available.

Alternatively, an initialization expression or value may be provided:

```
int *ip;            // ip is a pointer to integers
ip = new int(5);    // Memory allocation for one integer, initial value of integer is 5
```

These operators may also be used with user defined data types:

```
struct ComplexT{          // A structure to define complex numbers
     float re , im;       // real and imaginary parts
};

ComplexT *cp = new ComplexT;    // cp is a pointer to ComplexT, memory alloc.
       :
 delete cp;                     // releasing the memory
```

To define dynamic arrays, the new[size_of_array] operator must be used. A dynamically allocated array may be deleted using operator delete[].

```
int *ipd = new int[10];                   // memory alloc. for 10 integers
for (int k=0; k<10; k++) ipd[k]= 0;       // clearing elements of the array
delete [] ipd;                            // releasing the memory
```

Function Declarations and Definitions

C++ uses a stricter type checking.
In function declarations (prototypes) the data types of the parameters must be included in the parentheses.

```
char grade (int, int, int);    // declaration

int main()
{
   :
}
char grade (int exam_1, int exam_2, int final_exam)     // definition
{
  :            // body of function
}
```

inline Functions (Macros)

In C, macros are defined by using the #define directive of the preprocessor.

In C++ macros are defined as normal functions. Here the keyword inline is inserted before the declaration of the function.

Remember the difference between normal functions and macros:

A normal **function** is placed in a separate section of code and a call to the function generates a jump to this section of code.

Before the jump the return address and arguments are saved in memory (usually in stack).

When the function has finished executing, return address and return value are taken from memory and control jumps back from the end of the function to the statement following the function call.

The advantage of this approach is that the same code can be called (executed) from many different places in the program. This makes it unnecessary to duplicate the function's code every time it is executed.

There is a disadvantage as well, however.

The function call itself, and the transfer of the arguments take some time. In a program with many function calls (especially inside loops), these times can add up and decrease the performance.

An **inline function** is defined using almost the same syntax as an ordinary function.

However, instead of placing the function's machine-language code in a separate location, the compiler simply inserts it into the location of the function call.

Using macros increases the size of the program. But macros run faster, because transfer of parameters and return address is not necessary.

```
inline int max (int i1, int i2){         // An inline function (macro)
    return(i1 > i2) ?  i1 : i2;          // it returns the greatest of two integers
}
```

Calls to the function are made in the normal way:

```
int j, k, l ;                  // Three integers are defined
……….                           // Some operations over k and l
j = max( k, l )                // inline function max is inserted
```

The decision to inline a function must be made with some care.

It's appropriate to inline a function when it is short, but not otherwise.

If a long or complex function is inlined, too much memory will be used and not much time will be saved.

Default Function Arguments

A programmer can give default values to parameters of a function. In calling of the function, if the arguments are not given, default values are used.

Example:

```
void f(char c, int i1=0, int i2=1)     // i1 and i2 have default values
{ ... }                                // Body of the function is not important
```

This function may be called in three different ways:

```
f('A',4,6);   // c='A', i1=4, i2=6
f('B',3);     // c='B', i1=3, i2=1
f('C');       // c='C', i1=0, i2=1
```

In calling a function argument must be given from left to right:

```
f('C', ,7);   //  ERROR! Third argument is given, but second is not.
```

While writing functions, default values of parameters must be given from right to left without skipping any parameter.

```
void f(char c='A', int i1, int i2=1)   // ERROR! i1 has been skipped
```

Default values must not be only constant values. They may also be expressions or function calls. `void f(char c, int i1=other_func())`

Overloading of function Names

C++ enables several functions of the *same name* to be defined as long as these functions have different sets of parameters (Numbers, types or the order of the parameters may be different).

The name and the parameter list build the *signature* of the function.

Example:

```
struct ComplexT{        // Structure for complex numbers
    float re, im;
};

void print (float value){   // print function for real numbers
   cout << "value= " << value << endl;
}

void print (ComplexT c){    // print function for complex numbers
   cout << "real= " << c.re << " im= " << c.im << endl;
}

void print (float value, char c){    // print function for real numbers and chars
   cout << "value= " << value << " c= " << c << endl;
}
```

```
int main()
{
   ComplexT z;
   z.re=0.5;
   z.im=1.2;
   print(z);
   print(4.2);
   print(2.5,'A');
   return 0;
}
```

See Example e23.cpp

Reference Operator (&)

This operator provides an alternative name for storage.

```
int i = 5;
int &j = i;    // j is a reference to i.     j and i both have the same address.
j++;           // i = 6
```

Actually there is only one integer memory cell with two names: i and j.
It is often used to pass parameters to a function by their references.

Call by reference:
Remember; in C parameters are passed to functions always by their *values*. To pass parameters by their *addresses* pointers should be used.
If we want, that the function can modify the original value of a parameter, then we must send its address to the function.

Example: (Call by value)

```
void calculate(int j) {  j = j * j / 2; }  // j can not be changed,  function is useless
int main()                    Call by value
{
   int i=5;
   calculate(i);   // i can not be modified
   return 0;
}
```

Solution with pointers (C Style):

```
void calculate(int *j) {
    *j = *j**j/2;          // Difficult to read and understand
}
int main()                    Call by address
{
  int i=5;
  calculate(&i);            // Address of i is sent
  return 0;
}
```

Here the symbol & is not reference operator, it is address operator.

Solution with references (C++ Style):

```
void calculate(int &j) {      // j is a reference to the coming argument, the same address
   j = j*j/2;                 // In the body j is used as a normal variable
}
                              Call by reference
int main( )
{
    int i=5;
    calculate(i);       // A normal function call. But instead of value address is sent
    return 0;
}
```

Another reason for passing parameters by their address is avoiding large data to be copied into stack.

Remember all arguments which sent to a function are copied into stack. This operation takes time and wastes memory.

To prevent the function from changing the parameter accidentally, we pass the argument as **constant reference** to the function.

```
struct Person{                              // A structure to define persons
    char name [40];                         // Name filed 40 bytes
    int reg_num;                            // Register number 4 bytes
};                                          // Total: 44 bytes

void print (const Person &k)                // k is constant reference parameter
{
    cout << "Name: " << k.name << endl;     // name to the screen
    cout << "Num: " << k.reg_num << endl;   // reg_num to the screen
}

int main(){
    Person ahmet;                           // ahmet is a variable of type Person
    strcpy(ahmet.name,"Ahmet Bilir");       // name = "Ahmet Bilir"
    ahmet.reg_num=324;                      // reg_num= 324
    print(ahmet);                           // Function call
    return 0;
}
```

Instead of 44 bytes only 4 bytes (address) are sent to the function.

Return by reference:

By default in C++, when a function returns a value: return *expression*;

expression is evaluated and its value is copied into stack. The calling function reads this value from stack and copies it into its variables.

An alternative to "return by value" is "return by reference", in which the value returned is not copied into stack.

One result of using "return by reference" is that the function which returns a parameter by reference can be used on the left side of an assignment statement.

Example: This function returns a reference to the largest element of an array.

```
int&  max(int a[], int length)         // Returns an integer reference
{
    int i=0;                           // indices of the largest element
    for (int j=0 ; j<length ; j++)
        if (a[j] > a[i])    i = j;
    return a[i];                       // returns reference to a[i]
}
int main()
{
   int array[ ] = {12, -54 , 0 , 123, 63};    // An array with 5 elements
   max(array,5) = 0;                          // write 0 over the largest element
   :
```

See Example e24.cpp
Notice the usage of const and cons_cast

To prevent the calling function from changing the return parameter accidentally, **const** qualifier can be used.

```
const int&  max( int a[ ], int length)      // Can not be used on the left side of an
{                                           // assignment statement
      int i=0;                              // indices of the largest element
      for (int j=0 ; j<length ; j++)
           if (a[j] > a[i])    i = j;
      return a[i];
}
```

This function can only be on right side of an assignment

```
int main()
{
     int array[ ] = {12, -54 , 0 , 123, 63};     // An array with 5 elements
     int largest;                                // A variable to hold the largest elem.
     largest = max(array,5);                     // find the largest element
     cout << "Largest element is " << largest << endl;
     return 0;
}
```

Never return a local variable by reference!

Since a function that uses "return by reference" returns an actual memory address, it is important that the variable in this memory location remains in existence after the function returns.
When a function returns, local variables go out of existence and their values are lost.

```
int&  f( )                  // Return by reference
{
     int i;                 // Local variable. Created in stack
     :
     return  i;             // ERROR! i does not exist anymore.
}
```

Local variables can be returned by their values

```
int  f( )                   // Return by value
{
     int i;                 // Local variable. Created in stack
     :
     return  i;             // OK.
}
```

Operator Overloading

In C++ it is also possible to overload the built-in C++ operators such as +, -, = and ++ so that they too invoke different functions, depending on their operands.

That is, the + in a+b will add the variables if a and b are integers, but will call a different function if a and b are variables of a user defined type.

Some rules:

• You can't overload operators that don't already exist in C++.

• You can not change numbers of operands. A binary operator (for example +) must always take two operands.

• You can not change the precedence of the operators.

  * comes always before +

Everything you can do with an overloaded operator you can also do with a function.

However, by making your listing more intuitive, overloaded operators make your programs easier to write, read, and maintain.

Operator overloading is mostly used with objects. We will discuss this topic later more in detail.

Writing functions for operators:

Functions of operators have the name operator and the symbol of the operator. For example the function for the operator + will have the name operator+ .

Example: Overloading of operator (+) to add complex numbers:

```
struct ComplexT{            // Structure for complex numbers
    float re,im;
};

// Function for overloading of operator (+) to add complex numbers
ComplexT operator+ (const ComplexT &v1, const ComplexT &v2){
   ComplexT result;                 // local result
    result.re = v1.re + v2.re;
    result.im = v1.im + v2.im;
    return result;
}

int main(){
    ComplexT c1, c2, c3;    // Three complex numbers
    c3 = c1 + c2;           // The function is called. c3 = operator+(c1,c2);
    return 0;
}
```

See Example e25.cpp