C++ As a Better C

Feza BUZLUCA
Istanbul Technical University
Computer Engineering Department
http://faculty.itu.edu.tr/buzluca
http://www.buzluca.info

Overview

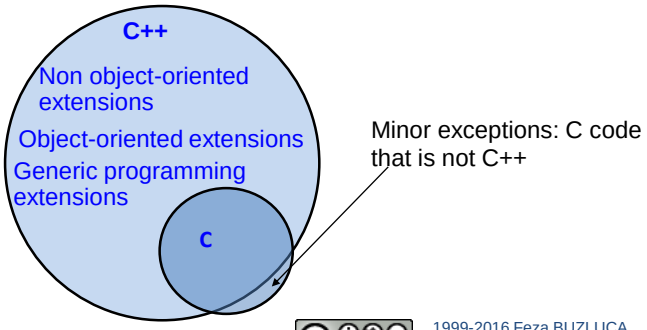
- C++ As a Better C
- C++ Enhancements to C
- Unary Scope Resolution Operator
- Namespaces
- Accessing Variables
- C++ Standard Library Headers
- Input/Output
- Bool Type
- Consts
- Casts
- Dynamic Memory Allocation
- Function Definitions and Declarations
- Inline Functions
- Default Arguments
- Function Overloading
- Reference Operator
- Pass-by-Reference
- Return by Reference
- Operator Overloading

C++ As a Better C

C++ was developed from C, by adding some features:

1. Non-object-oriented features programmers can use in the coding phase (These are not related to 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++ Enhancements to C

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 the 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

C++ Enhancements to C

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
```


C++ Enhancements to C

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 says: "Create this variable here" or "Create this function here." It allocates storage for the name.

C++ Enhancements to C

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 appear at the beginning of a block.
- In C++, declarations and definitions can be placed anywhere an executable statement can appear, but they must appear prior to the point at which they are first used.
  - This improves the readability of the program.

C++ Enhancements to C

- 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 beginning of for loop
   a++;
   int p = 12;                    // Definition of p
   ...                            // Scope of p
}                                 // End of scope for i and p
```

Unary Scope Resolution 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 if we use 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
}
```


Unary Scope Resolution Operator (::)

- Caution: Avoid giving identical names to global and local data, if possible.
- In C++ (as in C), an operator may have more than one meaning.
- The scope resolution operator has also many different tasks, which will be presented in following chapters.

Name Conflicts

- When a program reaches a certain size, it is typically broken up into pieces, each of which is built and maintained by a different person or group
- Since C effectively has a single area 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 arises if a programmer tries to use the same names as the names of library functions

Namespaces

- Standard C++ has a mechanism to prevent this collision: 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

Namespace 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 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

Accessing Variables: "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' was 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>
```

```
#include <iostream>
```

- 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 putting a "c" before the name.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <cstdio>
#include <cstdlib>
```

Std Namespace

- 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;
```


Header Files

- Today, most C++ compilers support old libraries and header files as well. So, you can also use the old header files with the extension '.h'.
- For a high-quality program, always use 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 I/O operations.
- When a C++ program includes the iostream header, four objects are created and initialized:
  - cin handles input from standard input (the keyboard)
  - cout handles output to standard output (the screen)
  - cerr handles unbuffered output to standard error device (the screen)
  - clog handles buffered error messages to 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 do not need std:: anymore
int main() {
   int i, j;                        // Two integers are defined
   cout << "Input two numbers \n";  // Message to screen,cursor to new line
   cin >> i >> j;                   // Read i and j from keyboard
   cout << "Sum= " << i + j << "\n";// Sum of numbers to screen
   return 0;
}
```

See Example e22.cpp

Bool Type

- bool type represents boolean (logical) values, for which 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.
- Since there is 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 use integers to produce logical values

Bool Type: Example

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

Constants

- In standard C, the preprocessor 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)
```

Constants

- const can appear before (left) and after (right) the type
  - both allowed and equivalent

```
int const MAX = 100;    // Same as const int MAX = 100;
```

- The keyword const is used very often in C++ programs, as we will see in this course. This usage decreases the possibility of error.
- To make your programs more readable, use uppercase letters for constant identifiers.

Const in Declaration of Pointers: Case 1

- The data pointed by the pointer is constant, but the pointer itself may be changed

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

- p is a pointer variable, which points to chars. We can also write “const” after the type:

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

- What p points to 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
```

Const in Declaration of Pointers: Case 2

- The pointer itself is a const pointer which may not be changed
- Data the pointer points to 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 constant)
```


Const in Declaration of Pointers: Case 3

- 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 const)
ssp++;                        // Compiler Error!(Because pointer 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 a floating point var
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

Casts: Static Cast

- static_cast<type>(expression) operator is used to convert one type to another acceptable 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
```

Casts: Const Cast

- const_cast<type>(expression) operator is used to do away with the const-ness of a (pointer) type
- Example:
  - p is a pointer to constant data
  - 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 programmer intends to make this assignment, then programmer must use const_cast operator:

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


Casts: Reinterpret Cast

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

Casts: Reinterpret Cast

- Example: 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 (2 x 4 = 8 bytes)
- x is a variable of type S
- Each byte of x can be reached using pointer xp

Casts: Dynamic Cast

- dynamic_cast<>() operator is used in the context of inheritance and polymorphism
- We will see inheritance and polymorphism later
- Discussion of this cast postponed until polymorphism

Casts: General Comments

- Using the cast operators is a dangerous habit, as it suppresses the normal type-checking mechanism of the compiler
- You should avoid casts if at all possible
- If casts have to be used, document well the reasons for their use in your code, to make 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 using standard library functions malloc and free
- C++ new and delete operators enable programs to perform dynamic memory allocation more easily
- Example:
  - 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
```

Dynamic Memory Allocation

- Note: new and delete are operators and therefore, do not require parentheses, as required for functions like malloc() and free()
- operator new returns a pointer to the kind of memory requested by its argument (e.g., a pointer to an int, in the previous example)
- Note: 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

Dynamic Memory Allocation

- 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{              // structure to define complex nums.
   float re, im;              // real and imaginary parts
};

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

Dynamic Memory Allocation

- To define dynamic arrays, 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;                   // initializing elements of the array
}
delete [] ipd;                   // releasing the memory
```


Function Declarations and Definitions

- C++ uses 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 exam1, int exam2, int finalExam)     // definition
{
  :             // body of function
}
```

Inline Functions

- In C, macros are defined 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

Difference Between a Normal Function and a Macro

- 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 the 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

Difference Between a Normal Function and a Macro

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

Inline Functions

- 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. However, 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;  // returns greatest of two ints
}
```

- 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
```

Inlining Functions

- The decision to inline a function must be made with some care
- It is appropriate to inline a function only when it is short
- Inlining a long or complex function uses too much memory and does not save much time

Functions: Default Arguments

- A programmer can give default values to parameters of a function
- When calling the function, if the arguments are not given, default values are used
- Example:

```
void f(char c, int i1 = 0, int i2 = 1)// i1, i2 have def. 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
```

Functions: Order of Arguments

- When calling a function, arguments 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 left to right without skipping any parameters

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

- Default values do not have to be only constant values
  - They may also be expressions or function calls

```
void f(char c, int i1 = other_func() )
```


Overloading 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 order of parameters may be different)
- Name and parameter list form the signature of the function

Overloading Function Names: Example

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

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

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

// print for real numbers and chars
void print(float value, char c){
   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 have same address
j++;        // i = 6
```

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

Pass-by-Reference vs. Pass-by-Value

- Remember: in C, parameters are always passed to functions by their values. To pass parameters by their addresses, pointers should be used.
- If we want the function to modify the original value of a parameter, then we must send the address of the parameter to the function.
- Example: Pass-by-value

```
// j can not be changed, function is useless
void calculate(int j) {  j = j * j / 2; }

int main()
{
   int i = 5;
   calculate(i);   // i can not be modified
   return 0;
}
```

Pass-by-value

Solutions

- Using pointers (C style)

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

Pass-by-address

Here, symbol & is not reference operator, but address operator

- Using references (C++ style)

```
void calculate(int &j) { // j reference to coming arg., same addr
    j = j * j / 2;       // In body, j is used as a normal var
}
int main()
{
    int i = 5;
    calculate(i); //normal func. call, but instead of value, addr sent
    return 0;
}
```

Pass-by-reference

Pass-by-Reference

- Another reason for passing parameters by their address is avoiding the copying of large data into stack
- Remember all arguments which are sent to a function are copied into stack. This operation takes time and wastes memory.

Passing Argument As a Const Ref.

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

```
struct Person {              // A structure to define persons
    char name[40];           // name field 40 bytes
    int id;                  // ID 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 << "ID: " << k.id << endl;         // id to the screen
}

int main(){
    Person ahmet;              // ahmet is variable of type Person
    strcpy(ahmet.name,"Ahmet Bilir");  // name = "Ahmet Bilir"
    ahmet.id = 324;                    // id = 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-hand side of an assignment statement

Return by Reference: 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;                        // index 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;                    // Set largest element to 0
     :
}
```

See Example e24.cpp
Note usage of const and const_cast

Return by Reference: Const

- To prevent the calling function from accidentally changing the return parameter, we can use the const qualifier

```
const int&  max( int a[ ], int length)  // Cannot be used on left-hand
                                        // side of an assignment
                                        // statement
{
   int i = 0;                           // index of 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};  // Array with 5 elements
   int largest;                      // Variable to store largest elem.
   largest = max(array, 5);          // find the largest element
   cout << "Largest element is " << largest << endl;
   return 0;
}
```


Never Return Local Variable by Reference!

- Function that uses “return by reference” returns an actual memory address
  - it is important that the variable in this memory location remain 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

Operator Overloading: Some Rules

- You cannot overload operators that do not already exist in C++
- You cannot change numbers of operands
  - A binary operator (for example, +) must always take two operands
- You cannot change the precedence of the operators
  - * comes always before +

Operator Overloading: Advantages

- 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 in more detail.

Writing Functions for Operators

- The function for an operator has the name operator, followed by the symbol of the operator
- For example, the function for the operator + will have the name operator+

Writing Functions for Operators

- 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