ISE103 – Programming II

Dr. Sanem Sarıel-Talay

C++ Programming Language

- C++ was developed from the C programming language, by adding some features to it. These features:
  - Non-object-oriented features, which can be used in coding phase.
  - Features which support object-oriented programming.
  - Features which support generic programming.
- With minor exceptions, C++ is a superset of C.

C++ Enhancements to C (Non-OO)

- 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++.
  - Always keep object-oriented and generic programming techniques in mind.
  - Always use C++ style coding technique
- Non object-oriented features of a C++ compiler can be also used in writing procedural programs.

C++ Enhancements to C (Non-OO)

- Comment Lines
  - /* This is a comment */
  - // This is a comment
- C++ allows you to begin a comment with // and use the remainder of the line for comment text.
  - This increases readability.

Declarations and Definitions in C++

- Declarations and definitions are different
- 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 allocates storage for the name
  - It says: "Make this variable here" or "Make this function here."

Declarations and Definitions in C++, con't

```
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)
```

Declarations and Definitions in C++, con't

- In C, declarations and definitions must be placed 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 improves the readability of the program.

Scope of a variable in 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 in the 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 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 ::
- Like in C, in C++ the same operator may have more than one meaning. The scope operator has also many different tasks.

```
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
}
```


Namespace

- A program is typically broken up into pieces
  - Each is built/maintained by a different person or group.
- In C, all the developers must be careful not to accidentally use the same names in situations where they can conflict.
  - Usage of the names of library functions.
- Standard C++ has a mechanism to prevent this collision: namespace. Each set of C++ definitions in a library or program is "wrapped" in a namespace
  - if some other definition has an identical name, but is in a different namespace -> no collision.

Namespace, con't

```
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
```


Namespace, con't

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

- By default, a variable or function is defined in the global namespace. It can be accessed without a namespace name and scope operator.
- using declaration makes it easier to access variables and functions, which are defined in a namespace.

```
using programmer1::iflag; //applies to a single-item
iflag = 3;                //  programmer1::iflag=3;
programmer2::iflag = -345;
programmer1::g(6);
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
- Through the evolution of C++, different extensions for file names (.hpp, .H , etc.) and different restrictions on file names, in particular on name length.
  - These issues caused source code portability problems.
- instead of #include <iostream.h>, you can now write: #include <iostream>

Standard C++ Header Files, con't

- 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 the 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.

I/O

- 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";// cursor to the new line
   cin >> i >> j;      // Read i and j from the keyboard
   cout << "Sum= " << i + j << "\n";
   return 0;
}
```

See Example e22.cpp

Bool type

- The type bool represents boolean (logical) values: true and false
- 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.

```
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, 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 = 5; // Compiler Error!
```

- 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;`
- Decreases error possibilities.
- To make your programs more readable, use uppercase font for constant identifiers.

Use of constants

The keyword const with the declaration of pointers

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

- `const char *p = "ABC";`

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

- `char const *p = "ABC";`

Whatever is pointed to by p may not be changed. The pointer p itself however may be changed.

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


Use of constants, con't

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
```


Use of constants, con't

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

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

- The same pointer definition may also be written as follows:

```
const char *const ssp = "ABC";
*ssp = 'Z'; // Compiler Error!
ssp++;   // Compiler Error!
```

- 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;
- 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;
float  f;
f = float(i)/2;
```

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

Casts – static_cast

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

```
int i=5;
float f;
f = static_cast<float>(i)/2;
```

Casts – 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?
```


Casts – 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);
   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.

Casts, con't

- 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.
- 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 C++

- 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.
-

```
in C:     int *p ;
          p = (int *) malloc(N*sizeof(int)) ;
          free(p) ;

in C++:   int *p ;
          p = new int[N] ; // allocation for an array
          delete []p ;  //allocated memory is released
                        for all elements of the array
```


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
}
```


Function Declarations and Definitions, con't

- In C++ a return type must be specified; a missing return type does not default to int as is the case in C.
- In C++, a function that has no parameters can have an empty parameter list.

```
int print (void);     /* C style  */
int print();         // C++ style
```

Hint

- In principle all C code could be recompiled as C++ .
- In practice, C++ has somewhat stricter type checking and the compiler is able to detect more bugs and issue more warnings.
- So recompiling C code using a C++ compiler is a way of cleaning up the existing code. The changes that have to be introduced into the source code at that stage are mostly bug fixes and stricter type enforcement.
- All this work is definitely worth the effort !

inline functions

- 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.

inline functions, con't

- When normal functions are used:
  - 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 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.
- But!! the function call itself, and the transfer of the arguments takes time. In a program with many function calls (especially inside loops), the performance may be degraded.

inline functions, con't

- `inline int max(int x,int y){return (y<x ? x : y); }`
- 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. :

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


inline functions, con't

- The decision to an inline a function must be made with some care:
  - If a function is more than a few lines long and is called many times, then inlining may require much more memory than an ordinary function.
  - 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.

```
int exp(int n,int k=2){
    if(k == 2)
      return (n*n) ;
    else
      return ( exp(n,k-1)*n ) ;
}
```

Default Function Arguments

- In calling a function, arguments must be given from left to right without skipping any parameter

```
void f(int i, int j=7); //right
void g(int i=3, int j); // wrong
void h(int i, int j=3, int k=7); // right
void m(int i=1, int j=2, int k=3); // right
void n(int i=2, int j, int k=3); // right ? wrong
```


Example

void n(int i=1, int j=2,int k=3) ;

- n() → n(1,2,3)
- n(2) → n(2,2,3)
- n(3,4) → n(3,4,3)
- n(5,6,7) →n(5,6,7)

Function Overloading

- 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.

```
double average(const double a[],int size);
double average(const int a[],int size);
double average(const int a[],const double b[],int size);
```


Function Overloading, con't

```
double average(const int a[],int size) { // Function (1)
   double sum = 0.0 ;
   for(int i=0;i<size;i++)
        sum += a[i] ;
   return (sum/size) ;
}
double average(const double a[],int size) {// Function (2)
   double sum = 0.0 ;
   for(int i=0;i<size;i++)
        sum += a[i] ;
   return (sum/size) ;
}
double average(const int a[],const double b[],int size) {
// Function (3)
   double sum = 0.0 ;
   for(int i=0;i<size;i++)
        sum += a[i] + b[i] ;
   return (sum/size) ;
}
```


Function Overloading, con't

```
int main() {
        int  w[5]={1,2,3,4,5} ;
        double  x[5]={1.1,2.2,3.3,4.4,5.5} ;
        cout << average(x,5) ; // Function (2)
        cout << average(w,5) ; // Function (1)
        cout << average(w,x,5) ; // Function (3)
        return 0 ;
}
```

See Example e23.cpp

Reference Operator - &

- This operator provides an alternative name for storage
- There are two usages of the operator

```
int n ;
int& nn = n ;
double a[5] ;
double& first = a[0] ;
double& last = a[4] ;
```

Function Call – Call by value

- If parameters are passed by value, we cannot modify the original value of a parameter

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

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

- To modify the values of the parameters, pointers should be used and the addresses of the parameters are passed to the function.

Function Call – Call by reference in C

```
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;
}
```

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

Function Call – Call by reference in C++

```
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
}

int main( ){
     int i=5;
     calculate(i); // A normal function call. But
                   //instead of value, address is sent
     return 0;
}
```


Function Call – Const Reference

- 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 screen
}

int main(){
    Person lost;    //lost is a variable of type Person
    strcpy(lost.name,"Jack Shephard"); //name = "Jack Shephard"
    lost.reg_num= 4815162342;          //reg_num= 4815162342
    print(lost);                       //Function call
    return 0;
}
```

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

Return by Reference, con't

- 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.

Return by Reference, con't

- 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.

```
int&  max( const 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};
   max(array,5) = 0;  // write 0 over the largest element
     :
```

See Example e24.cpp

Const Return Parameter

- 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;
        for (int j=0 ; j<length ; j++)
             if (a[j] > a[i])    i = j;
        return a[i];
}
```

- This function can only be on the right side of an assignment

```
int main(){
  int array[ ] = {12, -54 , 0 , 123, 63};
     int largest;
     largest = max(array,5);      // find the largest element
     cout << "Largest element is " << largest << endl;
     return 0;
}
```


Caution - returning local variables

- Since a 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: Rules

- You can't overload operators that don't already exist in C++.
- You can not change the 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.

Operator Overloading, con't

- 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+:

```
struct SComplex{
 float real,img;
};

SComplex operator+(SComplex v1, SComplex v2){
  SComplex result;
  result.real=v1.real+v2.real;
  result.img=v1.img+v2.img;
  return result;
}

int main(){
  SComplex c1={1,2},c2 ={5,1};
  SComplex c3;
  c3=c1+c2;   // c1+(c2)
}
```

See Example e25.cpp