

Lecture 2

C++ Features

1

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

C++ Language

- C++ is based on the C programming language.
- All C commands and built-in functions works also in C++.
- C++ contains features to support followings:
 - Object oriented programming
 - Generic programming (with templates)
- C++ language versions are published by ISO (International Organization for Standardization).

3

Advantages of C++ Language

- Speed of programs written with C++ is high.
- C++ supports object-orientation and generic programming (function templates and class templates).
- C++ has many standard built-in program libraries.
- C++ programmers can easily adapt to other object oriented programming languages (Java, C#, etc).

4

Application Areas of C++

- **Banking, trading, insurance:**
Maintainability, ease of extension, reliability.
- **Systems programming:**
Operating systems, device drivers.
Direct usage of hardware under real-time constraints.
- **Graphical User Interface (GUI) programs**
- **Computer communication / networking programs**

5

C++ Language Keywords

- The C/C++ keywords below are reserved.
- They can not be used as identifiers (such as variable names, constant names, class names, or function names).

auto	delete	goto	public	this
break	do	if	register	template
case	double	inline	return	typedef
catch	else	int	short	union
char	enum	long	signed	unsigned
class	extern	new	sizeof	virtual
const	float	overload	static	void
continue	for	private	struct	volatile
default	friend	protected	switch	while

6

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

For statement with a loop counter

- The i variable below is declared as a loop counter.
- Scope of the i is valid only within the for statement block.

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

int main()
{
    int A[4] = {10, 20, 30, 40 };

    for (int i=0; i < 4; i++)
        cout << A[i] << endl;
}
```

Screen Output

```
10
20
30
40
```

Range-based For statement in C++

- C++ language has the Range-based For statement, as an alternative method.
- The num variable below is declared as an item in the array, not as a loop counter.
- The colon operator ":" represents the Range operator.

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

int main()
{
    int A[4] = {10, 20, 30, 40 };

    for (int num : A)
        cout << num << endl;
}
```

Screen Output
(Same)

```
10
20
30
40
```

9

Built-in **string** class in C++

- A C++ **string** is a variable-sized character array.

```
#include <iostream> //For console input / output stream statements
using namespace std; //Standard namespace is used
int main() {
    // Define two string variables
    string s1;
    string s2 = "hello";
    cout << s1.size() << " " << s2.size() << endl;
    // Number of characters: 0 and 5

    s1 = s2 + ' ' + "world"; // Concatenation
    if (s1 == "hello world") // Comparison
        cout << "\nStrings are equal \n";

    cout << s1[0] << endl; // Displays first character in s1 → 'h'
    //s1.substr(m, n); → Substring of size n starting at s1[m]
    cout << s1.substr(2, 4) << endl;

    const char * ps;
    ps = s1.c_str(); // Convert string to const char*
    cout << ps << endl;

    cout << "Enter a long sentence : \n";
    getline(cin, s2); // Read line ending with newline
    cout << s2 << endl;
}
```

Screen Output

```
0 5
Strings are equal
h
llo
hello world
Enter a long sentence :
abc defgh ijklmn
abc defgh ijklmn
```

10

Initializations and **auto** type in C++

- Objects (variables) can be initialized with data in various ways.
- The **auto** type specifier makes the **compiler** automatically deduce type of a variable at compile-time, based on the initializer data type that was written.

```
#include <iostream>
using namespace std;
int main() {
    int x = 5;
    int y { 5 };
    int z = { 5 };

    auto x2 = 'a';    // char type
    auto y2 = 7;      // int type
    auto z2 = 3.8;    // double type

    cout << x << "\t" << y << "\t" << z
         << "\t" << x2 << "\t" << y2
         << "\t" << z2 << endl;
}
```

Screen
Output

5	5	5	a	7	3.8
---	---	---	---	---	-----

11

Declarations and Definitions

- A **declaration** introduces a name (an identifier) to the compiler. It does not allocate memory storage.
- A **definition** tells the compiler to make a variable/function. It allocates memory storage for the name.
- In C, declarations and definitions must occur at the beginning of a block.
- In C++, declarations and definitions can be placed anywhere prior to first used place.

12

Example: Declarations and Definitions

Declarations

```
// Declaration of structure  
// for complex numbers  
struct ComplexT  
{  
    float re;    // Real part  
    float im;    // Imaginary part  
};
```

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

Definitions

```
// Definition of a variable  
int j;
```

```
// Definition of object variables  
ComplexT c1, c2;
```

13

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

Block Scope

- A variable is available only in a code block, in which it was defined.
- The block is the **scope** of that variable.
- Blocks are written between brace paranthesis { and }.

Example :

```
int sum=0;
for (int i=0; i < 100; i++)
{ // Variable i is defined at the beginning of for loop block
    sum++;
    int x=12; // Definition of local x variable
              // Scope of x is limited to loop block
} // End of scope for i and x

cout << "i = " << i << endl;
cout << "x = " << x << endl; }
```

Compiler errors
(i and x are undefined outside of loop)

15

Scope Operator (::)

- A definition in a block (local name) can hide a definition in an enclosing (outer) block or a global name.
- A global name can be accessed by using the scope operator ::

```
int x=1; // Global x
int y=0; // Global y

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
}
```

It is recommended to give different names to global variables and local variables.

16

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

Namespaces

- A long program can be broken up into parts.
- Parts can be in the same file or in different files, maintained by different programmers.
- In C, the programmers must be careful not to use the same names for variables and functions in a project.
- C++ has a mechanism to prevent **name collision**:
The **namespace** keyword.
- If some other definition has an identical name, but is in a different namespace, then there is no collision.
- If a variable or function does not belong to any namespace, then it is defined in the **global namespace**.

Example: Namespaces

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

namespace programmer1 {           // programmer1's namespace
    int flag;                     // programmer1's flag
    void g (int);                 // programmer1's g function
}                                 // end of namespace

namespace programmer2 {           // programmer2's namespace
    int flag;                     // programmer2's flag
}

int main() {
    programmer1 :: flag = 3;      // programmer1's flag
    programmer2 :: flag = -5;    // programmer2's flag
    programmer1 :: g (6);        // programmer1's g function
}
```

19

The using declaration

The "**using**" declaration statement makes it easier to access variables and functions, which are defined in a **namespace**.

Example1: (Using only a selected item of namespace)

```
int main() {
    using programmer1 :: flag;
    // Applies to one item in the namespace
    flag = 3; // programmer1 :: flag = 3;
}
```

Example2: (Using all items of namespace)

```
int main() {
    using namespace programmer1;
    // Applies to all items in namespace
    flag = 3; // programmer1 :: flag = 3;
    g (6);    // programmer1's function g
}
```

20

The **std::numbers** namespace

- C++20 provides some of the mathematical constants.
- The <numbers> header file should be included.
- The constants themselves are inside the **std::numbers** namespace.

Compiling in
command-line

```
g++ -std=c++20 prog.cpp -o prog.exe
```

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

int main()
{
    cout << "pi          = " << pi          << endl;
    cout << "e           = " << e           << endl;
    cout << "log2(e)      = " << log2e       << endl;
    cout << "log10(e)     = " << log10e      << endl;
    cout << "ln(2)        = " << ln2         << endl;
    cout << "ln(10)       = " << ln10        << endl;
    cout << "sqrt(2)      = " << sqrt2       << endl;
    cout << "sqrt(3)      = " << sqrt3       << endl;
}
```

Screen Output

```
pi          = 3.14159
e           = 2.71828
log2(e)     = 1.4427
log10(e)    = 0.434294
ln(2)       = 0.693147
ln(10)      = 2.30259
sqrt(2)     = 1.41421
sqrt(3)     = 1.73205
```

21

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

Standard C++ header files

- C++ eliminates the file extension for header files.
- For example, instead of `#include <iostream.h>` you can write: `#include <iostream>`
- You can also use header files by putting “c” letter before the name.
 - `#include <stdio.h>` becomes `#include <cstdio>`
 - `#include <stdlib.h>` becomes `#include <cstdlib>`
- The libraries that have been inherited from C are still available with the traditional ‘.h’ extension.

23

The **std** Namespace

- In standard C++ header files, all declarations and definitions take place in a namespace called **std**.
- **std** is the abbreviation of “standard”.
- The followings are common beginning statements in a C++ program.
`#include <iostream>`
`using namespace std;`
- Some objects that the **std** namespace contains:
 - **cin** (console input)
 - **cout** (console output)
 - **endl** (end line)
 - **string**

24

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

Console Input / Output

- Instead of C library functions (**printf**, **scanf**), in C++ the library objects (**cout**, **cin**) are preferred to use for console I/O operations.
- When a C++ program includes the **iostream** header, the following I/O stream objects are initialized.
 - **cin** handles standard input (keyboard).
 - **cout** handles standard output (screen).

The **cout** object

- To print a value to screen, write the word **cout**, followed by the **<<** output operator.
- **Method1:** With the **using namespace std;** statement.
(Input/output statements can be written without the **std::** prefix.)

```
#include <iostream> //Header file for the cout object
using namespace std;
int main() {
    int i = 5;
    cout << "Integer number = " << i << endl;
}
```

- **Method2:** Without the **using namespace std;** statement.
(Input/output statements must be written with the **std::** prefix.)

```
#include <iostream> //Header file for the cout object
int main() {
    int i = 5;
    std :: cout << "Integer number = " << i << std :: endl;
}
```

27

The **cin** object

- To read data from keyboard, the predefined **cin** object is written with the **>>** input operator.
- **Method1:** With the **using namespace std;** statement.
(Input/output statements can be written without the **std::** prefix.)

```
#include <iostream>
using namespace std;
int main() {
    int i, j;
    cout << "Give two numbers : ";
    cin >> i >> j; // Read i and j from keyboard
    cout << "Sum= " << i + j << "\n";
}
```

- **Method2:** Without the **using namespace std;** statement.
(Input/output statements must be written with the **std::** prefix.)

```
#include <iostream>
int main() {
    int i, j;
    cout << "Give two numbers : ";
    std :: cin >> i >> j; // Read i and j from keyboard
    std :: cout << "Sum= " << i + j << "\n";
}
```

28

Example1 : Reading one char from keyboard

- The program below reads only one character from keyboard.
- User must press the ENTER key.

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

int main()
{
    char k;

    cout << "Enter one character : ";
    k = cin . get ();    // Same as k = getchar();
    //cin . get(k);      // Alternative method

    cout << k << endl;
}
```

**Screen
Output**

```
Enter one character : a
a
```

29

Example2 : Reading whole sentence from keyboard

- The program below reads a whole line from keyboard into a character array, until user presses the ENTER key.

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

int main()
{
    char s [50];

    cout << "Enter a sentence : ";
    cin . getline (s, 50);
    cout << s << endl;
}
```

**Screen
Output**

```
Enter a sentence : aaa bbb ccc
aaa bbb ccc
```

30

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

File Output in C++

- The program below creates a sequential data file (text file, not binary file).

```
#include <iostream> //Console Input/Output
#include <fstream> //File stream input/output
using namespace std;

int main() {
    // Output File Stream constructor opens file as output
    ofstream dosya ("veri.txt", ios::out);

    // Stop program if unable to create file
    if ( ! dosya ) {
        cout << "File could not be opened" << endl;
        return 0;
    }

    for (int i = 1; i <= 10; i++)
        dosya << i << " " << i*i << endl;

    cout << "Dosya olusturuldu\n";
    dosya.close();
}
```

OUTPUT file :
veri.txt

1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

File Input in C++

- The program below reads numbers from an existing sequential data file.

```
#include <iostream> //Console Input/Output
#include <fstream> //File stream input/output
using namespace std;

int main()
{
    int num1, num2;
    // Input File Stream constructor opens file
    ifstream dosya;
    dosya.open("veri.txt");

    if (! dosya.is_open()) {
        cout << "File can not be opened\n";
        return 0;
    }

    while ( ! dosya.eof() )
    {
        dosya >> num1 >> num2;
        cout << num1 << " " << num2 << endl;
    }

    dosya.close();
}
```

INPUT file :
veri.txt

1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

33

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

Formatting Stream Output with **<iostream>** formatters

- Format specifiers can be used to generate a formatted output stream.
- A formatter is applied to an output stream with the insert operator, <<.
- They are defined in **<iostream> header file** ([Input/Output Stream](#)).

Formatter	Explanation
fixed	Output floating-point data in fixed-point notation.
scientific	Output all subsequent floating-point data in scientific notation, which always includes an exponent and one digit before the decimal point.
defaultfloat	Revert to the default floating-point data presentation.
dec, hex, oct	All subsequent integer output is decimal, hexadecimal, octal, respectively.
showbase	Outputs the base prefix for hexadecimal and octal integer values. Inserting <code>std::noshowbase</code> in a stream will switch this off.
left, right	Output is left-justified, right-justified (default) in the field.

35

Formatting Stream Output with **<iomanip>** formatter functions

- The **<iomanip> header file** ([Input/Output Manipulation](#)) provides parametric formatter functions.
- When a formatter is used, it normally remains in effect until you change it.
- The only exception is `std::setw()`, which only influences the width of the next field that is output.

Formatter Function	Explanation
setprecision (n)	<ul style="list-style-type: none"> • Sets the floating-point precision or the number of decimal places to n digits. • If the default floating-point output presentation is in effect, n specifies the number of digits in the output value. • If fixed or scientific format has been set, n is the number of digits following the decimal point. • The default precision is 6.
setw (n)	<ul style="list-style-type: none"> • Sets the output field width to n characters, but only for the next output data item. • Subsequent output reverts to the default where the field width is set to the number of output character needed to accommodate the data.
setfill (ch)	<ul style="list-style-type: none"> • Sets the fill character to be ch for all subsequent output. • When the field width has more characters than the output value, excess characters in the field will be the default fill character, which is a space.

36

Example1 : Formatting integer number outputs

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

int main()
{
    int a = 23, b = 78;

    cout << setw(5) << a << setw(5) << b << endl;

    cout << left << setw(5) << a << setw(5) << b << endl;

    cout << " a = " << setbase(16) << setw(6) << showbase << a
        << " b = " << setw(6) << b << endl;

    cout << setw(10) << a << setw(10) << b << endl;
}
```

**Screen
Output**

```
    23    78
23    78
a = 0x17    b = 0x4e
0x17        0x4e
```

37

Example2 : Formatting float number outputs

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

int main()
{
    float num = 9.5;

    cout << fixed
        << setw(6)
        << setprecision(2)
        << setfill('0')
        << num;
}
```

Screen Output

```
009.50
```

38

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- **Bool type**
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

The **bool** type

- The data type **bool** represents boolean (logical) values.
- A logical value can be *true* or *false*.
- The compiler considers nonzero values as **true**, zero values as **false**.

```
bool is_greater;           // Boolean variable: is_greater
is_greater = false;        // Assigning a logical value
int a=8, b=3;
.....
is_greater = a > b;         // Logical operation

if (is_greater) .....     // Conditional operation
```

Type Aliases

- A type alias is a user-provided name for the existing data type.
- To use a different name for an existing data type, write the following:
`using my_type_name = existing_type_name;`

C Syntax

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

int main() {
    // Type definition in C syntax.
    typedef int    MyInt;
    typedef string MyString;
    MyInt x    = 123;
    MyString s = "Hello World";
    cout << x << " " << s << endl;
}
```

C++ Syntax

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

int main() {
    // Type definition in C++ syntax.
    using MyInt    = int;
    using MyString = string;
    MyInt x    = 123;
    MyString s = "Hello World";
    cout << x << " " << s << endl;
}
```

41

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

Defining Constants

- In C, directive `#define` is used to define constants.
`#define MAX 100`
- In C++, a named constant can be defined just like a variable, except that its value cannot be changed.

```
const int MAX = 100;    // MAX is constant.  
MAX = 5;    // Compiler Error (MAX can not be changed)
```

- The `const` word can be written before or after the type.
`int const MAX = 100;`
- Uppercase letters are preferred for defining constants.

43

const pointers

The keyword `const` can be used in declaration of pointers.
There are three different cases:

Example1: The data pointed by the pointer is constant.
But the pointer itself may be changed.

```
const char *p = "ABC"; // Constant data  
  
*p = 'Z';    // Compiler error: Because data is constant.  
  
p++;        // Address in the pointer may change.
```

44

const pointers

Example2: The pointer itself is constant, which may not be changed.
Value pointed may be changed.

```
char * const sp = new char [5]; // Constant pointer  
  
strcpy (sp, "ABC");  
  
*sp = 'Z'; // Data changed to "ZBC"  
  
sp++; // Compiler error: Because pointer is constant
```

Example3: Both the pointer and the pointed data are constants.

```
const char * const ssp = "ABC"; // Constant pointer and data  
  
*ssp = 'Z'; // Compiler error: Because data is constant  
  
ssp++; // Compiler error: Because pointer is const
```

45

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

Type Casting

(Type conversions)

C offers the following cast method to convert a data type to another type.
(typename) expression.

C-style type casting example:

```
int i=7;
float f;
f = (float) i / 2; // Convert i to float, then divide
```

C++ offers the following type casting methods.

All are template based operators (template functions).

- `static_cast`
- `const_cast`
- `reinterpret_cast`
- `dynamic_cast`
(related to inheritance)
- `any_cast`

47

The static_cast

- The `static_cast<type>(expression)` operator is used to convert one type to other type.
- The `< >` syntax is used for template parameter.

```
int i=7;
float f;
```

// Variable i is converted to float, then divided by 2.

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

↙
Name of
operator
(function)

↓
Float is
template
parameter

↘
i is
function
argument

48

The const_cast

The `const_cast<type>(expression)` operator can be used for constant type casting.

Example:

- p is a pointer to constant data.
- q is a pointer to non-constant data.
- 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

q = const_cast <char *> (p);

*q = 'X'; // Compiler does not give error.
           // But this command causes run-time error
```

49

The reinterpret_cast

The `reinterpret_cast<type>(expression)` operator is used to reinterpret the bytes in a variable.

Example: The individual bytes making up a structure can be reached using a `reinterpret_cast`.

```
struct S { // Structure with total 8 bytes
    int i1, i2;
};

int main() {
    S a; // a is of type S
    a.i1 = 2000;
    a.i2 = 3000;
    unsigned char *p; // Pointer to unsigned chars (bytes)
    p = reinterpret_cast <unsigned char *> ( &a );

    // Display the bytes of struct
    for (int j=0; j<8; j++) // Count up to 8 bytes
        cout << static_cast <int> ( p [j] ) << endl;
} // end of main
```

50

The any_cast

The `any_cast <type> (expression)` operator is used for the conversion of generic data type named the `any`.

```
#include <iostream>
#include <any> //Container header file
using namespace std;

int main()
{
    any a; //Variable a is generic data type.

    a = 200;
    cout << any_cast <int> (a) << endl;

    a = string ("Hello");
    cout << any_cast <string> (a) << endl;

    a = float (3.14);
    cout << any_cast <float> (a) << endl;
}
```

Screen Output

```
200
Hello
3.14
```

51

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

Dynamic Memory Allocation

- In C++ , the **new** and **delete** operators are used for dynamic memory allocation.

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

- An **initialization** value may also be provided.

```
int * p;           // ip is a pointer to integer
p = new int (-50); // Memory allocation for one integer,
                  // initial value of integer is -50
```

53

Example1: Dynamic Memory Allocation (For array)

To define dynamic arrays, the **new [size_of_array]** operator is used.

```
int * pd;
pd = new int [10]; // memory allocation for 10 integers

for (int k=0; k<10; k++)
    pd [k] = 2*k;   // setting elements of the array

delete [] pd;       // releasing the whole array memory
```

54

Example2: Dynamic Memory Allocation (For struct)

The **new** and **delete** operators may also be used with user defined data types such as a struct.

```
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
.....
delete cp;    // releasing the memory
```

55

inline Functions

- In C++, an **inline function** is defined using the same syntax as an ordinary function.
- Compiler makes a copy and paste of inline function codes, every time it is called.
- Inline functions run faster.

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

inline int MAX (int i1, int i2)
{
    if (i1 > i2) return i1;
    else return i2;
}

int main() {
    int a=5, b=8;
    cout << MAX (a,b) << endl;
}
```

56

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

Default Function Arguments

- Default values to parameters (arguments) can be written in a function.
- In calling of the function, if the arguments are not given, default values are used.

```
// n1 and n2 have default values
void f (char k, int n1 = 0, int n2 = 8)
{
    cout << k << ' ' << n1 << ' ' << n2 << endl;
}
```

The function may be called in three different ways:

```
f ('A', 4, 6); // k='A', n1=4, n2=6
f ('A', 4);   // k='A', n1=4, n2=8
f ('A');      // k='A', n1=0, n2=8
f ();         // Compiler error (k is null)
```

Default Function Arguments

- When calling a function, arguments must be given from **left-to-right**:
`f ('A', ,7);` // **ERROR:** *Third argument is given, but second is not.*
- While writing a function declaration, default values of parameters must be written from **right-to-left** without skipping any parameter.

```
void f (char k='A', int n1, int n2=8) {...}  
// ERROR: n1 has been skipped
```

Valid
declarations:

```
void f (char k,    int n1,    int n2=8);  
void f (char k,    int n1=0,  int n2=8);  
void f (char k='A', int n1=0,  int n2=8);
```

59

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- **Overloading of Function Names**
- Function Parameter Passing Methods
- Lambda Functions

Overloading of Function Names

- C++ enables several functions of the **same name** to be defined.
- Functions should have different sets of parameters.
- The name and the parameter list describe the *signature* (prototype) of the function.

```
struct ComplexT
{
    float re, im;
};

int main() {
    ComplexT z;

    z.re = 0.5;
    z.im = 1.2;

    print(z); //complex
    print(8); //integer
}
```

```
void print(int val)
{
    // print function for int numbers
    cout << "Value= " << val << endl;
}
```

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

61

Reference Operator (&)

The reference operator (&) provides an alternative name for memory storage.

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

j is an **alias name** of i.

```
j++; // i = 6
```

62

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

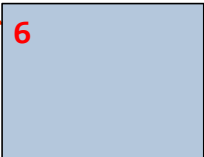
Function Parameter Passing Methods

- Parameters can be passed to functions by three methods.
- **Call-by-Value** (default)
- **Call-by-Address** (uses pointer)
- **Call-by-Reference** (only in C++, similar to call-by-address)
- If we want that the function can **modify** the original value of a parameter, then we must send its address/reference to the function.

Call-by-Value Method

Call by value

```
void calculate (int j)  
{  
    j = (j * j) / 2; // j is local, so change will be locally only.  
}  
  
int main()  
{  
    int i = 6;  
    calculate (i); // i can not be modified by function.  
    cout << i;  
}
```




Screen output
(i is not changed)

65

Call-by-Address Method

Call by address

```
void calculate (int *j) {  
    *j = (*j) * (*j) / 2;  
    // Modifies data which is pointed by j pointer.  
    // *j in function body means the content of location pointed by j.  
}  
  
int main()  
{  
    int i = 6;  
    calculate (&i); // Address of i is sent, i can be modified by function.  
    cout << i;  
}
```



Screen output
(i is changed)


66

Call-by-Reference Method

Call by reference

```
void calculate (int &j)    // j is a reference to the argument.
                          // Reference is same as an adress.
{
    j = (j*j) / 2;        // j is used as a normal variable
}
```

```
int main( ) {
    int i = 6;
    calculate (i);        // A normal function call.
                          // Instead of value, address is sent.
                          // i can be modified by function.
    cout << i;
```



Screen output
(i is changed)

67

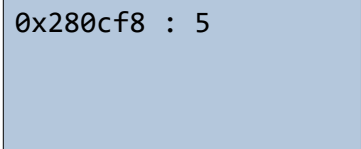
Return by Address

The function below returns a pointer to int, which is dynamically allocated.

```
int * f () {            // Return type is pointer-to-int
    int * x;            // Pointer definition
    x = new int;        // Dynamically memory allocation of an integer
    *x = 5;            // Data is initialized
    return x;          // x is an adress
}
```

```
int main()
{
    int * p;            // Pointer definition
    p = f ();           // Calling the function
    cout << p << " : " << *p << endl;
    // Displays the address and the value (5)
}
```

Screen output



0x280cf8 : 5

68

Constant Reference

To prevent a function from changing a parameter, we should pass the argument as **constant reference**, by using the ampersand & reference symbol.

```
struct Person {           // Total of 44 bytes
    char name [40];       // name   : 40 bytes
    int num;              // number : 4 bytes
};

void print (const Person & k) {
    // k is a constant reference parameter
    cout << "Name : " << k.name << endl;
    cout << "Number : " << k.num << endl;
    // k.num = 7513;      // Compiler error
                        // Because k is constant
}

int main() {
    Person x;             // x is a variable of type Person
    strcpy (x.name, "ABCD");
    x.num = 7513;
    print (x);            // Function call
}
```

Instead of 44 bytes (data),
only 4 bytes (reference of x)
is sent to the **print()** function.

69

Topics

- C++ Language
- For Statements
- Scope Operator
- Namespaces
- Standard C++ header files
- Console Input/Output statements
- File Input/Output statements
- Stream Output formatters
- Bool type
- Constants
- Type casting
- Dynamic Memory Allocation
- Default Function Arguments
- Overloading of Function Names
- Function Parameter Passing Methods
- Lambda Functions

Lambda Functions (Anonymous Functions)

- A lambda function (expression) is an anonymous function object.
- A **function object** is an object of a class.
- A function object can be called, similar to an ordinary function.
- The difference between an ordinary function and a lambda function:
A lambda function can be called as an object.
- They are mostly used in STL built-in functions.
- **General syntax** for defining a lambda function is below.
- The capture variables and parameter variables are optional, so that they can be omitted.
- The syntax "->" is optional, it describes the returned data type.

```
[ capture variables ] ( parameter variables ) -> returned_data_type  
{  
    code statements  
};
```

71

Example : Defining and calling a lambda function

- Program below defines a lambda function and calls.
- x and y are function parameters.
- The function's return type is auto (determined automatically by compiler.)

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    auto toplu = [] (int x, int y) { return x + y; };  
    cout << "Sonuc1 : " << toplu(6, 4) << endl;  
  
    // Define two function objects f1 and f2 and call them.  
    auto f1 = toplu, f2 = toplu;  
    cout << "Sonuc2 : " << f1(30, 20) << endl;  
    cout << "Sonuc3 : " << f2(15, 60) << endl;  
}
```

Screen
Output

```
Sonuc1 : 10  
Sonuc2 : 50  
Sonuc3 : 75
```

72