PUCIT, University of the Punjab

DEPARTMENT OF COMPUTER SCIENCE

COURSE OUTLINE

Course Title	Programming Fundamentals
Credit Hours	3+1
Semester	Spring 2020
Follow up	Object Oriented Programming
Instructor	Natalia Chaudhry

Description

Computer programming is an art of developing computational solutions to precisely describable problems. The purpose of this course is to introduce students with basic concepts of structured programming. After completing this course, they should be able to write elegant structured programs to solve different computational problems. Programs are demonstrated using the C++ programming language. However, the concepts are taught in a language independent fashion. Note that the basic purpose of this course is to learn programming instead of a particular programming language. The following topics will be covered in this course: Introduction to Programming Languages and Compilers; Flowcharts, Pseudo-code; Data Storage; Introduction to C++, Classes, Objects, and UML; Control Statements; Functions and Recursion; Debugging; Command Line Arguments; Preprocessor; Arrays; Pointers and Pointer-Based Strings; File Processing; Structures and Unions; Self-Referential Classes.

Learning Outcomes

1. Understand and use the fundamental concepts of data types, structured programming, algorithmic design, and user interface design.
2. Demonstrate a fundamental understanding of software development methodologies, including modular design, pseudo code, flowcharting, structure charts, data types, control structures, functions, and arrays.
3. Develop projects that utilize logical algorithms from specifications and requirements statements.
4. Demonstrate appropriate design, coding, testing, and documenting of computer programs that implement project specifications and requirements.
5. Apply computer programming concepts to new problems or situations..

Recommended Books

- *Deitel & Deitel, C++ - How to Program 5th Edition, Pearson – Prentice Hall (2005),*
- Walter Savitch, Problem Solving with C++: The Object of Programming, Addison-Wesley, ISBN-10: 0321268652
- Robert Lafore, Object-Oriented Programming in C++, ISBN-10: 0672323087.
- Gaddis - Starting Out with C++ - From Control Structures through Objects (6th Edition) BRIEF

Teaching Methodology

- Lectures
- Homework
- Quizzes
- Midterms
- Final
- Assignments
- Project
- Lab work

Grading criteria

Student's performance will be evaluated through the following criteria:

Assignments	5%
Quizzes	10%
Midterms	35%
Final Examination	40%
Project	10%
Total	**100%**

Tentative Plan

Week	Topics to cover
1	**Introduction:** Motivation, Rules for the course **Problem Solving:** Algorithm Design, Flow charts and idea of pseudocode **Intro to C++ Program Structure:** Introduction of C++, (Compilation, Linking, Loading, Execution & Debugging), Problem Analysis and Solution Design, C++ Basic Programs, Basic Program Structure, Data types, Variables declaration
2	What is Programming, Different Programming Errors, Variables, Primitive Data Types in C++, sizeof operator, typedef Declaration, Primitive Data types, Char data type, Operators in C++ , Assignment Rules, Operator precedence Constants & Literals (Types and ways to define constants), Expression and Its types, Evaluation of a given expression, Escape Sequence their uses, Quadratic Equation, Use of different operators in Expressions, Input Stream and its member functions, Use of operators, Type Casting (Conversion) in C++ and its different forms, Type Promotion Logic Structure (Flow), Sequential Structure, Decision making in C++, Selection Statements (if structure), Understating if structure with examples, use of Relational Operators in if structure, Flow Charts and its use in writing a good program, Logical Operators and their use in if Structure.
3	Decision Making Continue, if/else structure, Understating with if/else with examples If/else..if Structure & its usage, Understanding if/else..if with examples, Nested if structure & its usage. Conditional or ternary Operator (?) , Switch Statement, Break & Continue Statement, goto Statement
4	Repetitive Structure, Loops Structure with examples, While Loop Structure, Understanding while loop with example, While loop variation with user input, computing sum of odd and even numbers, infinite loop Do-while loop structure, while and do-while loop comparison, increment & decrement operators
5	For loop structure, understanding for loop with examples, nested loops, nesting different loops, Functions, Structure of Functions, Function Declaration and Definition, Writing your first function in C++, Example programs for understanding function , Math functions in C++
6	Scope of variables, Functions Arguments, Mixed arguments, Function call by reference and call by value, Argument Promotion Rules, inline functions, return a reference from function, dangling reference Function overloading, Function Template, static & Register Variables,
7	Recursion, Recursive Functions, Understanding recursive function call with example, header files in C++, File Inclusion, Conditional File Inclusion Arrays in C++, Usage of Arrays, Declaring an array, Types of Arrays. Arrays with example, copying Arrays
8	Searching & Sorting Techniques on Arrays, Linear Search Sorting and Searching Arrays examples
9	Arrays manipulation and Using Character Array to store & Manipulate String

	Static Local Array and Automatic Local Array, Functions and Arrays, Passing Arrays to functions, Multidimensional Arrays
10	Pointers, Pointers & Addresses, pointers & Variables, Pointers and Functions arguments, Changing the case of string with pointers in a function call, constant and non-constant pointers and data manipulation with them Pointers with arrays, sorting with pointers, Address Arithmetic, character pointers & functions, pointer arrays; pointer to pointer, pointers vs multidimensional array, command line arguments
11	Relationship between pointers & Arrays, pointer Expression & Arithmetic, pointers Comparison, Pointes, String & Arrays, String handling, String Manipulation Functions, Character handling, String Conversion Functions
12	Files, Text File handling, Output File handling, Example programs Sequential Access File, Random Access File, Data Insertion Techniques,
13	Basics of Structure, Structures & Functions, Arrays of Structure Pointers to Structure, self-referential Structure, union variable
14	Memory Allocation, Dynamic memory Allocation, memory leak, dangling pointers
15	Recursion
16	Revision

Programming fundamentals

WEEK 1

Outline

- Memory
- Need for algorithms
- Program vs algorithm vs flochart vs pseduocode
- Low level vs high level languages
- Role of assembler and compiler
- Structured programming vs object oriented programming (brief idea)
- Programming languages
- Integrated development environment (IDE)
- Two basic steps in software design/ problem solving
  - Analysis phase (algorithm designing, logic building)
  - Implementation phase (designing C++ programs using algorithm)
- Six steps in software development in C++ (yet to be discussed completely)
  - Edit
  - Preprocess
  - Compile
  - Link
  - Load
  - Execute

Recommended reading (not mandatory for now)

Deital & Deital, 9th edition

Page 9: Topic 1.5: Machine Languages, Assembly Languages and High-Level Languages

Walter Savitch, Problem Solving with C++ The Object of Programming

Page 1-9: Introduction - Compilers

D.S.Malik - C++ Programming - From Problem Analysis to Program Design (5th Edition)

Page 8: Topic: The Evolution of Programming Languages

Page 20: Programming Methodologies

Practice tasks..

Design algorithm and flowchart

- Calculate average and percentage of 5 subjects' marks
- Convert hours to minutes
- Convert weeks to hours
- Convert yards to meters
- Convert centigrade to Fahrenheit
- Convert Fahrenheit to centigrade
- Find area of rectangle
- Find area of triangle
- Find perimeter of shapes
- Display sum of squared of two numbers
- Display squared and cubes of 5 numbers.
- Input a no. check if it is positive or negative
- Input a no. Check it is even or odd?
- Input two numbers (numerator and denominator). For division, check if denominator is zero, print divide by zero error or print answer of division.
- Input a no. and check whether it is in range 10-70 or not.
- Input record of a student (name, age, address) along with marks of 5 subjects. And if age is valid print percentage of marks.
- Print maximum of two numbers
- Print maximum of three numbers.
- Input 3 numbers. Print sum of two largest numbers.

Programming Fundamentals

LECTURE: 13-2-2020

Recommended reads

Deital & Deital , 9 th edition

Page 17-19: *Topic 1.9:* **Typical C++ Development Environment**

Walter Savitch, Problem Solving with C++ The Object of Programming

Page 24-25: Layout of a simple C++ program

Lafore - Object-Oriented Programming in C++ (4th Edition)

Page 35: Directives, Header files, The using directive

D.S.Malik -C++ Programming -From Problem Analysis to Program Design (5th Edition)

Page 10-14: Processing a C++ Program

Outline of discussed topics

Functions

- Inputs (parameters/arguments), output
- definition/declaration
- Function calls

Library files:

- Header files
- Implementation files

Source code vs object code vs executable code vs library

6 steps of program development in C

Design first C++ program

Next lecture agenda

- **Recap:** Flowchart design practice and ***algorithm tracing***
- Start with designing programs in C++
- Basic syntax

PROGRAMMING FUNDAMENTALS

WEEK 2: *17-2-2020 , 19-2-2020*

AGENDA

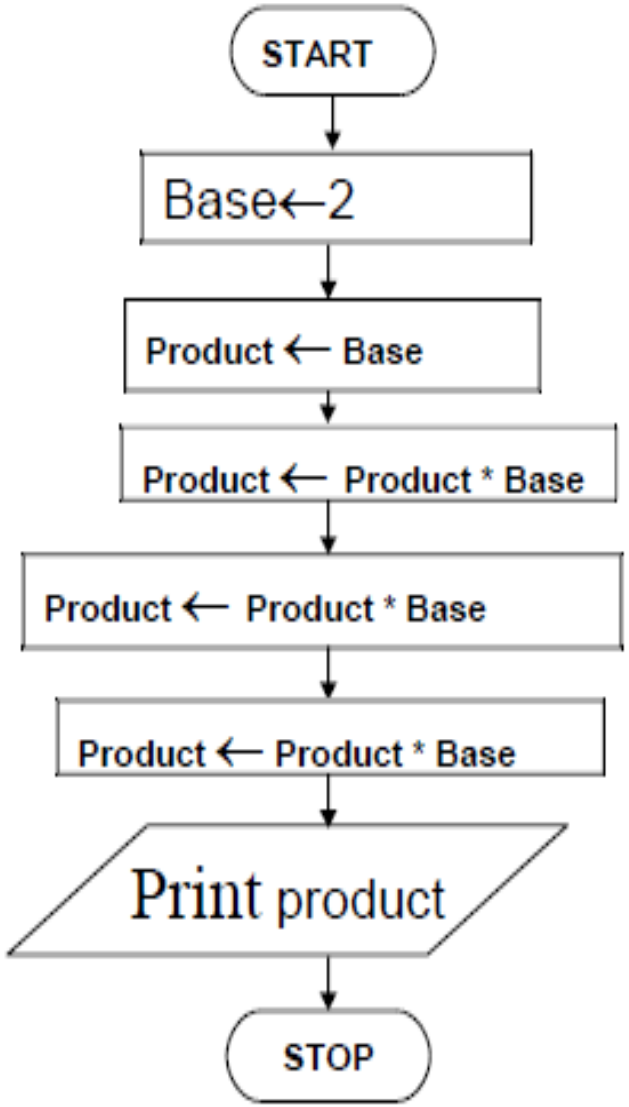
- Recap of previous week
- Algorithm tracing and problem solving
- Simple program designing in c++

RECAP

- Computers and problem solving
- Memory
  - RAM (volatile)
  - Files stored on disk/hard drive
- File types vary according to the nature of data
  - File extension
- Code/program in terms of Algorithm + flowchart
- Binary data and need for high level languages
- Syntax rules of languages, Compilers, and IDEs
- Six steps involved in program design
  - Edit
    - Source code (.cpp) vs object code (.obj) vs executable file (.exe)
    - Source code may comprise of **header file (.h)** and **implementation file (.cpp)**
  - Preprocess
  - Compile
  - Link
  - Load
  - Execute

ALGORITHM TRACING AND PROBLEM SOLVING

- **Example 1:** Write an algorithm and draw a flowchart to calculate 2^4. Verify your result by dry run.
- Algorithm:
- **Step 1:** Base = 2
- **Step 2:** Product = Base
- **Step 3:** Product = Product * Base
- **Step 4:** Product = Product * Base
- **Step 5:** Product = Product * Base
- **Step 6:** Print Product


START
Base←2
Product ← Base
Product ← Product * Base
Product ← Product * Base
Product ← Product * Base
Print product
STOP

Repeated execution of the same set of instructions

- Algorithm:
- Step 1: **Base** = 2
- Step 2: **Power** = 4
- Step 3: **counter** = 1, **product**= 1
- Step 4: **Product** = **Product** * **Base**
- Step 5: **counter** = **counter** + 1
- Step 6: if **counter** IS LESS THAN **Power**
    - Goto Step 4
    - Else
    - Goto Step 7
- Step 7: Print Product

ALGORITHM TRACING/DRY RUN

- Running the algorithm stepwise
- For each step, note the value of variables and conditions side by side
- Repeat this, till the program execution ends.

Step #	Base	Power	Counter	Product	Counter<Power
1	2	?	?	?	?
2	2	4	?	?	?
3	2	4	0	1	?
4	2	4	0	2	?
5	2	4	1	2	?
6	2	4	1	2	1<4,T
4	2	4	1	4	?
5	2	4	2	4	?
6	2	4	2	4	2<4,T
4	2	4	2	8	?
5	2	4	3	8	?
6	2	4	3	8	3<4,T
4	2	4	3	16	?
5	2	4	4	16	?
6	2	4	3	16	4<4, F
7	2	4	3	16	?
Print 16					

- **Algorithm:**
- Step 1: **Base** = 2
- Step 2: **Power** = 4
- Step 3: **counter** = 0, **Product=1**
- Step 4: **Product** = **Product** * **Base**
- Step 5: **counter** = **counter** + 1
- Step 6: if **counter** IS LESS THAN **Power**
    - Goto Step 4
    - Else
    - Goto Step 7
- Step 7: Print Product

- **Example 1:** Write down an algorithm to find and print the largest of three numbers. Read numbers one by one.
- Step 1: Input N1
- Step 2: Max = N1
- Step 3: Input N2
- Step 4: If (N2>Max)
    - Goto Step 5

  Else
    - Goto Step 6
- Step 5: Max = N2
- Step 6: Input N3
- Step 7: If (N3>Max)
    - Goto Step 8

  Else
    - Goto Step 9
- Step 8: Max = N3
- Step 9: Print “The largest number is:”,Max

Step #	N1	N2	N3	Max	N2>Max	N3>Max
1	5	?	?	?	?	?
2	5	?	?	5	?	?
3	5	6	?	5	?	?
4	5	6	?	5	6>5,T	?
5	5	6	?	6	?	?
6	5	6	1	6	?	?
7	5	6	1	6	?	1>6, F
9	5	6	1	6	?	?
Print “The largest number is:”,Max						

- Step 1: Input N1
- Step 2: Max = N1
- Step 3: Input N2
- **Step 4: If (N2>Max)**
    - **Goto Step 5**

  Else
    - **Goto Step 6**
- Step 5: Max = N2
- Step 6: Input N3
- **Step 7: If (N3>Max)**
    - **Goto Step 8**

  Else
    - **Goto Step 9**
- Step 8: Max = N3
- Step 9: Print “The largest number is:”,Max

- **Example 1:** Write down an algorithm to find and print the largest of N numbers.

1. Input N
2. count=0
3. Max=-1
4. Input a number n
5. If max < n

    goto Step 6

   Else

    goto Step 7
6. max=n
7. count=count+1
8. If count < N

    goto Step 4

   Else

    goto Step 9
9. print max

C++ PROGRAM DESIGN

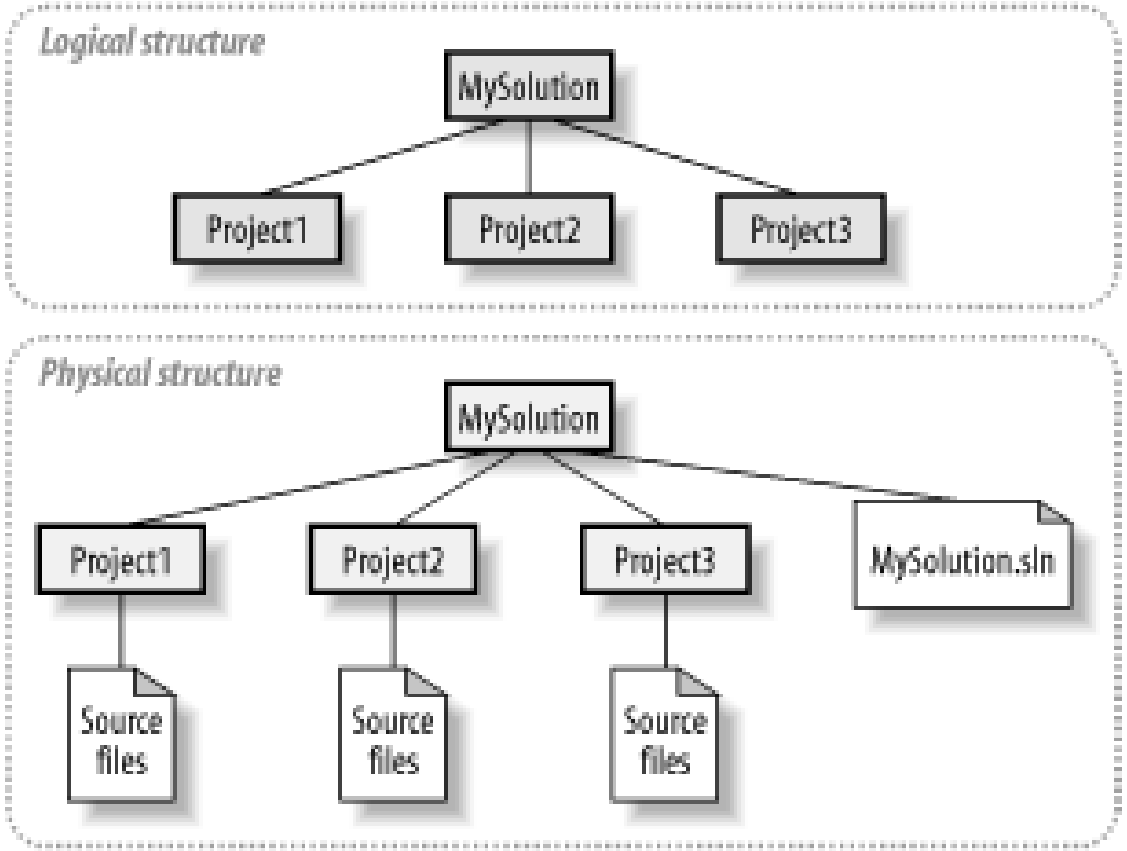
- Basic program structure
- Include directive and creating c++ file
- Stream insertion *Cin* and extraction *cout*
- Practice cout using **escape sequence**
- Writing Comments
- Memory concepts and variables
- Practice cin using **variables**

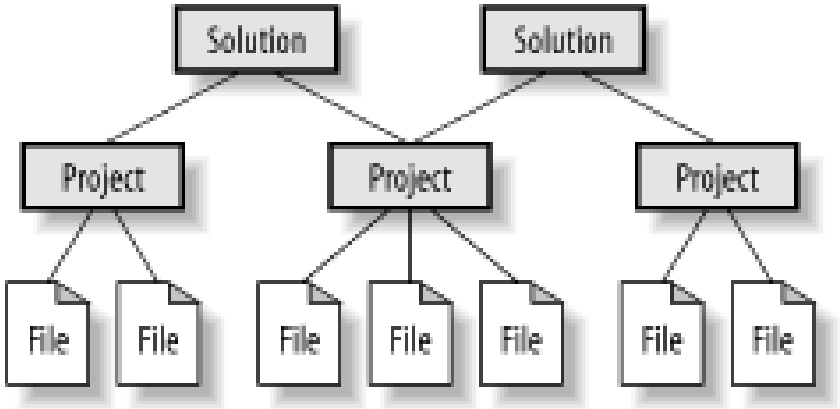
Solution vs project in MS Visual studio

- A project has two main jobs: to act as a container for our source files and to compile those files into some kind of component, typically either a Dynamic Link Library (DLL) or Windows Executable (EXE).

- A solution contains a collection of projects, along with information on dependencies between those projects.

- A solution is like a house, a project like a room. Each room provides a piece of functionality whereas the house, a container of rooms, provides the means to link the rooms together and organize them appropriately.


Logical structure
MySolution
Project1
Project2
Project3
Physical structure
MySolution
Project1
Project2
Project3
MySolution.sln
Source files
Source files
Source files


Solution
Solution
Project
Project
Project
File
File
File
File
File
File
File

BASIC PROGRAM STRUCTURE

```
#include <iostream>
using namespace std;
int main()
{
        cout << "Welcome ";
        return 0;
}
```

Escape sequences

`\n`	Newline. Position the screen cursor to the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
`\a`	Alert. Sound the system bell.
`\\`	Backslash. Used to print a backslash character.
`\'`	Single quote. Used to print a single quote character.
`\"`	Double quote. Used to print a double quote character.

PRINTING MULTIPLE LINES (USING ESCAPE SEQUENCES)

```
#include <iostream>
using namespace std;
int main()
{
        cout << "Welcome ";
        cout << "to C++!\n";
        return 0;
}
```

PRINTING MULTIPLE LINES (USING SINGLE COUT)

```
#include <iostream>
using namespace std;
int main()
{
        cout << "Welcome to\nC++!\n";
        return 0;
}
```

VARIABLES

```
#include <iostream>
using namespace std;
int main()
{
        int var1;
        cin>>var1;
        cout << var1;
        return 0;
}
```

Programming fundamentals

Lecture 4: Formatting output & Data types

Recap

- Cin and cout
- Variables declaration/definition
- Assignment operator and variable initialization
- I/O in c++ basics

Formatting Output: Manipulators

- Stream Manipulators are functions specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators on stream objects
- **Setw, setfill, setprecision**
- **#include <iomanip>** must be used

setw

- **The setw( ) manipulator** sets the width of the field assigned for the output.
- cout << setw(6) <<"R“;
- Output: _ _ _ _ _R (each underscore represents a blank space)
- How to right justify numbers?
- cout << setw(8) << 22 << "\n";
  cout << setw(8) << 4444 << "\n";
  cout << setw(8) << 666666 << endl;

setw

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    cout << "without setw :";
    cout << "Hello" << endl;

    cout << "setw(10):";

    cout <<setw(10) << endl;
    cout << "Hello" << endl;
    cout << "F19" << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
without setw:Hello
setw(7):
     Hello
F19
```

setw( ) manipulator **does not stick** from one cout statement to the next

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout << setw(10);
    cout << "Hello" << endl;
    cout << setw(10);
    cout << "F19" << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
     Hello
       F19
```

Padding field with characters of your own choice

setfill character is used in output operations to fill spaces when results have to be padded to the field width.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout << setfill('0');
    cout << setw(6);
    cout << "F19" << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
000F19
```

setprecision( )

- used to **set the number of decimal places to be displayed**

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout << setprecision(6);
    cout << 123.456789 << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
123.457
```

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout << setprecision(2);
    cout << 123.456 << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
1.2e+02
```

Setf()

- cout.setf(ios**::argument**);
- Test the output by changing the argument with:

left	left-justify the output
right	right-justify the output
scientific	display floating point numbers in scientific ("E") notation
fixed	display floating point numbers in normal notation - no trailing zeroes and no scientific notation

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout.setf(ios::right);
    cout <<setw(10)<< 123.456 << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
123.456
```

Integer data types

- Integer data types hold whole numbers. C++ has eight different integer data types.
- They differ based on how large of an integer they can store **(how many bytes are allocated for storage)** and whether they can store both positive and negative integers, or only nonnegative integers.

Data Type	Size*	Range
int	4 bytes	-2,147,483,648 to +2,147,483,647
unsigned int	4 bytes	0 to +4,294,967,295
short	2 bytes	-32,768 to +32,767
unsigned short	2 bytes	0 to +65,535
long long	8 bytes	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
unsigned long long	8 bytes	0 to +8,446,744,073,709,551,615

Integer: int

- It takes 4 bytes
- 4 bytes= 4x8=32 bits => 2^32= 4,294,967,296 i.e. **(0** to **4,294,967,295)**
- Integers consists of both negative and positive numbers. Thus we need to divide that number by 2.
- 4,294,967,296/2 =2,147,483,648
- Thus we have negative numbers in range from -1 to -2,147,483,648
- And positive numbers have range from 0 to 2,147,483,647 (not 2,147,483,648 because we start positive numbers from 0)

				MIN						MAX			
Input value	--	--	-2,147,483,649	-2,147,483,648	-2,147,483,64	--	0	--	2,147,483,646	2,147,483,647	2,147,483,648	--	--
Output on console	Max -2	Max-1	Max 2,147,483,647	-2,147,483,648	-2,147,483,64	-- -	0	-- -	2,147,483,646	2,147,483,647	Min -2,147,483,648	Min+1	Min+2

			MIN						MAX			
--	--	-2,147,483,649	-2,147,483,648	-2,147,483,64	--	0	--	2,147,483,646	2,147,483,647	2,147,483,648	--	--
Max-2	Max-1	Max 2,147,483,647	-2,147,483,648	-2,147,483,64	-- -	0	-- -	2,147,483,646	2,147,483,647	Min -2,147,483,648	Min+1	Min+2

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    int x;
    x = -2147483647-1;
    cout << x;
    return 0;
```

Microsoft Visual Studio Debug Console

```
-2147483648
```

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
int x;
x = 2147483648;
cout << x;
return 0;
}
```

Microsoft Visual Studio Debug Console

```
-2147483648
```

Checking the amount of memory allocated

- Exact size depends on architecture of computer (e.g. 32 and 64 bit systems)
- You can check the exact size using **sizeof** operator

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int myIntVariable = 1000;
    // use sizeof with types
    cout << "bytes for short: " << sizeof(short) << endl;
    cout << "bytes for int: " << sizeof(int) << endl;
    // use sizeof with variable
    cout << "bytes for myIntVariable: " << sizeof(myIntVariable) <<
    endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
bytes for short: 2
bytes for int: 4
bytes for myIntVariable: 4
```

Float and double

Floating point data types are used to store real numbers (numbers that can have a fractional part). There are three C++ floating point data types, float, double, and long double.

Data Type	Size*
float	4 bytes
double	8 bytes
long double	8 bytes

```
#include <iostream>
using namespace std;
int main()
{
float num = 4.5;
cout << num;
return 0;
}
```

char

- The char data type is used for single characters.
- A char literal is a single character enclosed in single quotes.

```
#include <iostream>
using namespace std;
int main()
{
    char alphabet =  'a';
    cout << alphabet;
    return 0;
}
```

string

- **#include <string>** must be used

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

int main()
{
    string myName = “Ali";  // string literals are enclosed in double
    quotes

    cout << "My name is: " << myName << endl;
    return 0;
}
```

bool

- The data type bool is for Boolean variables. Boolean data are either true or false.

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

int main()
{
bool myBooleanVariable = false;  // initialize as false

myBooleanVariable = true; // switch value to true

cout << "Value of my variable is : " << myBooleanVariable << endl;
return 0;
}
```

Programming fundamentals

Lecture 5: Data types, operators, conditional statements

Recap

- Formatting output: setfill, setw, setprecision

Agenda

- Data types,
- operators, and
- conditional statements

Setf()

- cout.setf(ios**::argument);**
- Test the output by changing the **argument** with:

left	left-justify the output
right	right-justify the output
scientific	display floating point numbers in scientific ("E") notation
fixed	display floating point numbers in normal notation - no trailing zeroes and no scientific notation

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout.setf(ios::right);
    cout <<setw(10)<< 123.456 << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
  123.456
```

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout.setf(ios::left);
    cout <<setw(10)<< 123.456 << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
123.456
```

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout.setf(ios::scientific);
    cout <<setw(10)<< 123.456 << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
1.234560e+02
```

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout.setf(ios::fixed);
    cout <<setw(10)<< 123.456 << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
123.456000
```

Data types

Integer data types

- Integer data types hold whole numbers. C++ has eight different integer data types.
- They differ based on how large of an integer they can store **(how many bytes are allocated for storage)** and whether they can store both positive and negative integers, or only nonnegative integers.

Data Type	**Size***	**Range**
int	4 bytes	-2,147,483,648 to +2,147,483,647
unsigned int	4 bytes	0 to +4,294,967,295
short	2 bytes	-32,768 to +32,767
unsigned short	2 bytes	0 to +65,535
long long	8 bytes	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
unsigned long long	8 bytes	0 to +8,446,744,073,709,551,615

Integer: int

- It takes 4 bytes
- 4 bytes= 4x8=32 bits => 2^32= 4,294,967,296 i.e. **(0** to **4,294,967,295)**
- Integers consists of both negative and positive numbers. Thus we need to divide that number by 2.
- 4,294,967,296/2 =2,147,483,648
- Thus we have negative numbers in range from -1 to -2,147,483,648
- And positive numbers have range from 0 to 2,147,483,647 (not 2,147,483,648 because we start positive numbers from 0)

				MIN						MAX			
Input value	--	--	-2,147,483,649	-2,147,483,648	-2,147,483,647	--	0	--	2,147,483,646	2,147,483,647	2,147,483,648	--	--
Output on console	Max -2	Max -1	Max 2,147,483,647	-2,147,483,648	-2,147,483,647	-- -	0	-- -	2,147,483,646	2,147,483,647	Min -2,147,483,648	Min+1	Min+2

			MIN						MAX			
--	--	-2,147,483,649	-2,147,483,648	-2,147,483,647	--	0	--	2,147,483,646	2,147,483,647	2,147,483,648	--	--
Max-2	Max-1	Max 2,147,483,647	-2,147,483,648	-2,147,483,647	--	0	--	2,147,483,646	2,147,483,647	Min -2,147,483,648	Min+1	Min+2

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    int x;
    x = -2147483647-1;
    cout << x;
    return 0;
```

Microsoft Visual Studio Debug Console

```
-2147483648
```

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
int x;
x = 2147483648;
cout << x;
return 0;
}
```

Microsoft Visual Studio Debug Console

```
-2147483648
```

Checking the amount of memory allocated

- Exact size depends on architecture of computer (e.g. 32 and 64 bit systems)
- You can check the exact size using **sizeof** operator

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int myIntVariable = 1000;
    // use sizeof with types
    cout << "bytes for short: " << sizeof(short) << endl;
    cout << "bytes for int: " << sizeof(int) << endl;
    // use sizeof with variable
    cout << "bytes for myIntVariable: " << sizeof(myIntVariable) << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
bytes for short: 2
bytes for int: 4
bytes for myIntVariable: 4
```

Float and double

Floating point data types are used to store real numbers (numbers that can have a fractional part). There are three C++ floating point data types, float, double, and long double.

Data Type	Size*
float	4 bytes
double	8 bytes
long double	8 bytes

```
#include <iostream>
using namespace std;
int main()
{
    float num = 4.5;
    cout << num;
    return 0;
}
```

char

- The char data type is used for single characters.
- Interpreted as integers, ASCII codes
- A char literal is a single character enclosed in single quotes.

```
#include <iostream>
using namespace std;
int main()
{
    char alphabet = 'a';
    cout << alphabet;
    return 0;
}
```

string

- **#include <string>** must be used

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

int main()
{
    string myName = “Ali";  // string literals are enclosed in double quotes

    cout << "My name is: " << myName << endl;
    return 0;
}
```

bool

- The data type bool is for Boolean variables. Boolean data are either true or false.
- 1 byte

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

int main()
{
bool myBooleanVariable = false;  // initialize as false

myBooleanVariable = true; // switch value to true

cout << "Value of my variable is: " << myBooleanVariable << endl;
return 0;
}
```

Operators

Operators

- An *operator* is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- *Arithmetic, assignment, relational* and *logical,* and *bitwise*

	Operator	Type
Unary operator	+ +, - -	Unary operator
Binary operator	+, -, *, /, %	Arithmetic operator
	<, <=, >, >=, ==, !=	Relational operator
	&&, ||, !	Logical operator
	&, |, <<, >>, ~, ^	Bitwise operator
	=, +=, -=, *=, /=, %=	Assignment operator
Ternary operator	?:	Ternary or conditional operator

Arithmetic

++ and – (increment and decrement operators

x = x+1;

can be written as

++x; // prefix form

or as

x++; // postfix form

Operator's precedence

highest	++ – –
	– (unary minus)
	* / %
lowest	+ –

Operator(s)	Operation(s)
( )	Parentheses
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction

e.g. solving expression

Algebra: $z = pr\%q + w/x - y$

C++: `z = p * r % q + w / x - y;`

6 1 2 4 3 5

e.g. same operator used multiple times

Step 1.	`y = 2 * 5 * 5 + 3 * 5 + 7;` `2 * 5 is 10`	*(Leftmost multiplication)*
Step 2.	`y = 10 * 5 + 3 * 5 + 7;` `10 * 5 is 50`	*(Leftmost multiplication)*
Step 3.	`y = 50 + 3 * 5 + 7;` `3 * 5 is 15`	*(Multiplication before addition)*
Step 4.	`y = 50 + 15 + 7;` `50 + 15 is 65`	*(Leftmost addition)*
Step 5.	`y = 65 + 7;` `65 + 7 is 72`	*(Last addition)*
Step 6.	`y = 72`	*(Last operation—place 72 in y)*

Relational and Logical Operators

Relational Operators	
Operator	**Meaning**
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
Logical Operators	
Operator	**Meaning**
&&	AND
||	OR
!	NOT

highest	!
	> >= < <=
	= = !=
	&&
lowest	||

Bitwise operators

- The **& (bitwise AND)** in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
- The **| (bitwise OR)** in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
- The **^ (bitwise XOR)** in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
- The **<< (left shift)** in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
- The **>> (right shift)** in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
- The **~ (bitwise NOT)** in C or C++ takes one number and inverts all bits of it

e.g

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int b = 9;
    cout << "b<<1: "<<(b<<1)<<endl;
    b = 9;
    cout <<"b<<2: "<< (b << 2)<<endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
b<<1: 18
b<<2: 36
```

Conditional statements

- If
- If, else
- If, else if,…., else

Programming fundamentals

Lecture 6: Operators

Recap

- Data types,
- operators, and
- conditional statements

Agenda

- Operators

Operators

- An *operator* is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- *Arithmetic, assignment, relational* and *logical,* and *bitwise*

	Operator	Type
Unary operator	+ +, - -	Unary operator
Binary operator	+, -, *, /, %	Arithmetic operator
	<, <=, >, >=, ==, !=	Relational operator
	&&, ||, !	Logical operator
	&, |, <<, >>, ~, ^	Bitwise operator
	=, +=, -=, *=, /=, %=	Assignment operator
Ternary operator	?:	Ternary or conditional operator

Unary

- The increment (++) and decrement (--) operators.

```
int a = 1;
 int b = a++;   // b = 1
 int c = a;     // c = 2
```

- The unary minus (-) operator. ***(changes the sign of its argument. A positive number becomes negative, and a negative number becomes positive.)***

```
int a = 10;
 int b = -a;  // b = -10
```

- The logical not (!) operator. ***(reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false)***

```
int a=2;
If(!(a==2)) {            }
```

Increment and decrement

++ and -- (increment and decrement operators

x = x+1;

can be written as

++x; // prefix form

or as

x++; // postfix form

Operator's precedence

highest	++ – –
	– (unary minus)
	* / %
lowest	+ –

Operator(s)	Operation(s)
( )	Parentheses
*	Multiplication
/	Division
%	Modulus
+	Addition
–	Subtraction

e.g. solving expression

Algebra: $z = pr\%q + w/x - y$

C++: `z = p * r % q + w / x - y;`

6 1 2 4 3 5

e.g. same operator used multiple times

Step 1.	`y = 2 * 5 * 5 + 3 * 5 + 7;` `2 * 5 is 10`	*(Leftmost multiplication)*
Step 2.	`y = 10 * 5 + 3 * 5 + 7;` `10 * 5 is 50`	*(Leftmost multiplication)*
Step 3.	`y = 50 + 3 * 5 + 7;` `3 * 5 is 15`	*(Multiplication before addition)*
Step 4.	`y = 50 + 15 + 7;` `50 + 15 is 65`	*(Leftmost addition)*
Step 5.	`y = 65 + 7;` `65 + 7 is 72`	*(Last addition)*
Step 6.	`y = 72`	*(Last operation—place 72 in y)*

Relational and Logical Operators

Relational Operators	
Operator	**Meaning**
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
Logical Operators	
Operator	**Meaning**
&&	AND
||	OR
!	NOT

highest	!
	> >= < <=
	= = !=
	&&
lowest	||

Bitwise operators

- The **& (bitwise AND)** in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
- The **| (bitwise OR)** in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
- The **^ (bitwise XOR)** in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
- The **<< (left shift)** in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
- The **>> (right shift)** in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
- The **~ (bitwise NOT)** in C or C++ takes one number and inverts all bits of it

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

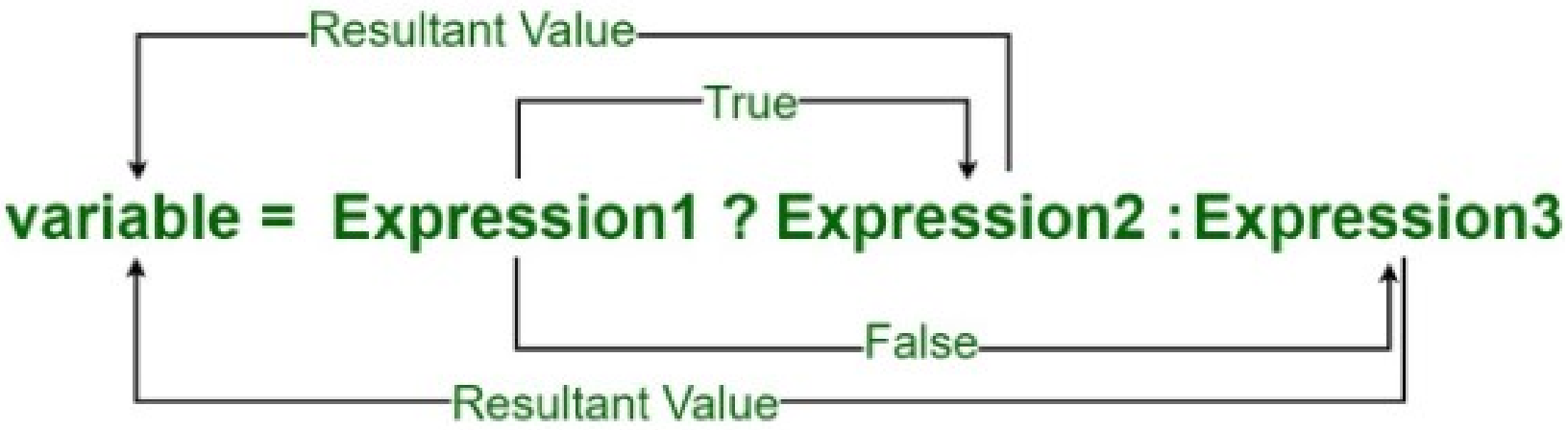
e.g

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int b = 9;
    cout << "b<<1: "<<(b<<1)<<endl;
    b = 9;
    cout <<"b<<2: "<< (b << 2)<<endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
b<<1: 18
b<<2: 36
```

Ternary operator (conditional operator) *? :*



Consider this code:

```
if ( a < b )
{
  a = b;
}
else
{
  a = -b;
}
```

You can replace the above code with:

```
a = (a < b) ? b : -b;
```

(a<b) is an if condition
b is a true case (if)
-b is a false case (else)

The ternary operator is more readable than a if...else statement for short conditions.

```
#include <iostream>
Using namespace std;
int main()
{
// variable declaration
int n1 = 5, n2 = 10, max;

// Largest among n1 and n2
max = (n1 > n2) ? n1 : n2;

// Print the largest number
Cout<<"Largest number between";
Cout<<n1<< n2<< max<<endl;

return 0;
}
```

If we want to print something

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    int n1 = 5, n2 = 10, max;

    // Largest among n1 and n2
    string m = (n1 > n2) ? "n1 is greater" : "n2 is greater";
    cout << m;
return 0;
}
```

Recommended reads

- Dietal & Dietal
  - Chapter 4
    - page 10: **section 4.5:** if selection statement, **4.6**, till page 114
    - Page 139: **Section 4.11:** Assignment operators, **4.12**
- Walter Savitch, Problem Solving with C++ The Object of Programming
  - Chapter 2
    - Page 60-82
- Schildt - C++ From the Ground Up (3rd Edition)
  - Chapter 3

Using multiple increment/decrement operators in one single expression

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
        int x = 6;
        int ans = 0;
        ans = ++x + ++x + ++x;
        cout << ans;
        return 0;
}
```

Output: 27

How this statement will be evaluated?

ans = ++x + ++x + ++x;

Keeping in mind the precedence of operators, unary operators (++ and --) have highest precedence. Thus, they will be evaluated first in any expression, then + will be evaluated. The other rule of precedence says that if we have same operator multiple times in a single expression, then it will be evaluated starting from left side and moving towards right.

Thus, three ++ operators will be evaluated one after the other, starting from the left side

ans = ++x + ++x + ++x;

7 8 9

At this point, the variable x got the final value "9". The equation then looks like:

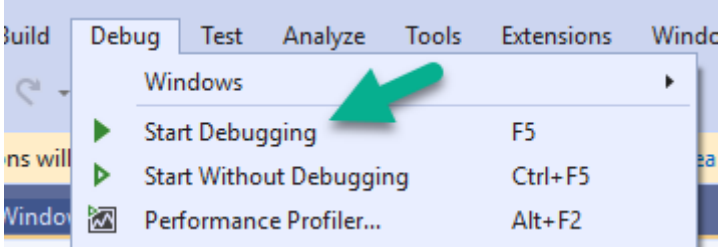
ans = x + x + x; //final value of x will be considered
9 + 9 + 9

ans= 27

Debugging guide v1

The process of solving code bugs is called Debugging. This basically means investigating the program's execution and state

Debugging in Visual Studio occurs automatically when you run from Visual Studio with F5 or select **Debug | Start Debugging**.



Break points:

You set breakpoints wherever you want to pause debugger execution. For example, you may want to see the state of code variables or look at the call stack at a certain breakpoint.

To set a breakpoint in source code, click in the far left margin next to a line of code. You can also select the line and press **F9**, select **Debug** > **Toggle Breakpoint**, or right-click and select **Breakpoint** > **Insert breakpoint**. The breakpoint appears as a red dot in the left margin.

To disable a breakpoint without deleting it, hover over or right-click it, and select **Disable breakpoint**. Disabled breakpoints appear as empty dots in the left margin or the **Breakpoints** window. To re-enable a breakpoint, hover over or right-click it, and select **Enable breakpoint**.

Steps to debug a simple code (basic level)

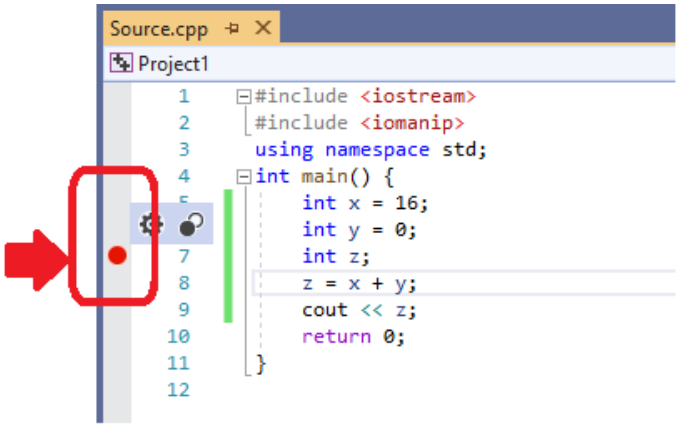
Step 1

Create a main cpp file (by creating new project) and paste the following code.

```
#include <iostream>
using namespace std;
int main() {
	int x = 16;
	int y = 0;
	int z;
	z = x + y;
	cout << z;
	return 0;
}
```

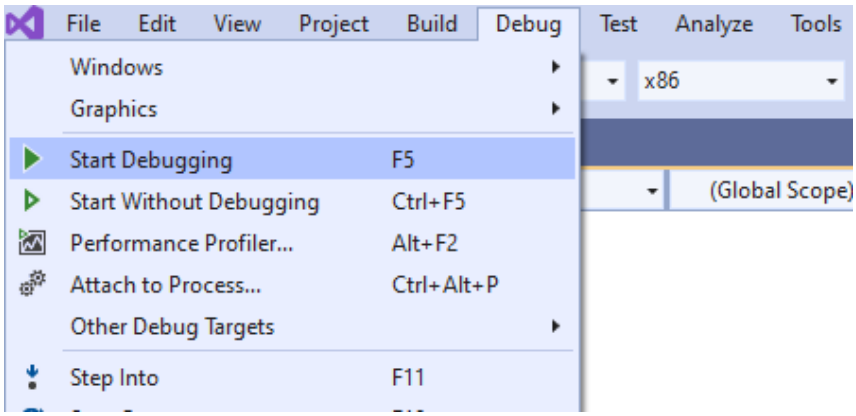
Step 2

You need to toggle a break point (according to the instructions mentioned above at the start). As you can see the following screenshot, you can clock on the left most side of the line to toggle a break point directly. Once toggled, red filled circle appears there.



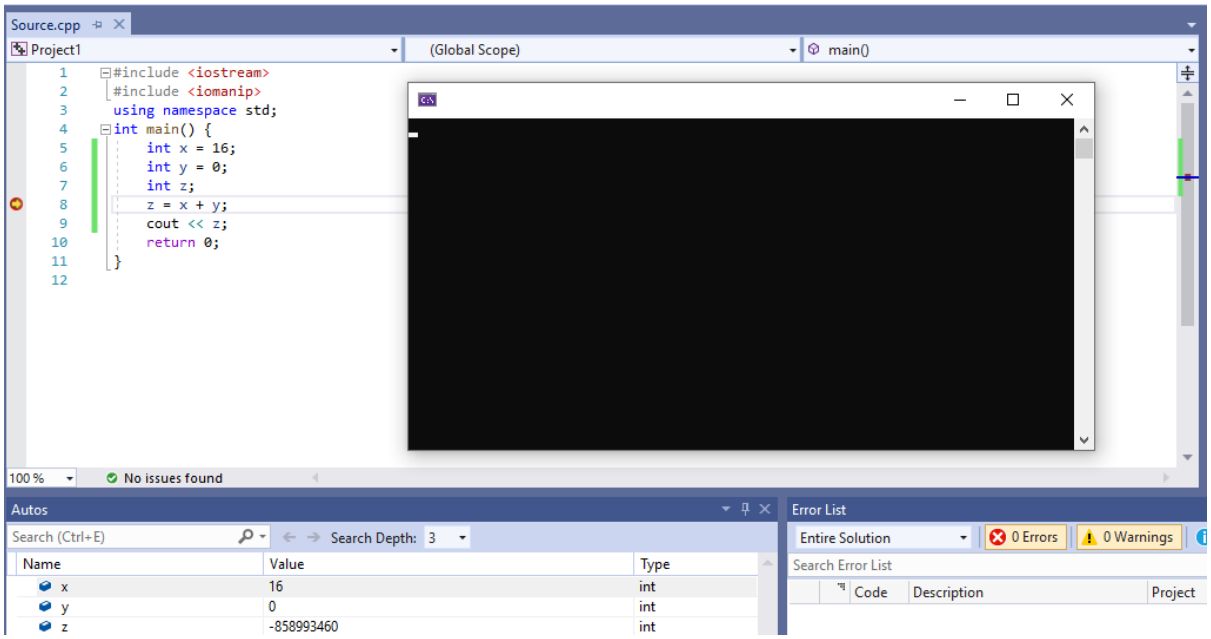
Step 3

Next move to **Debug-> Start Debugging**

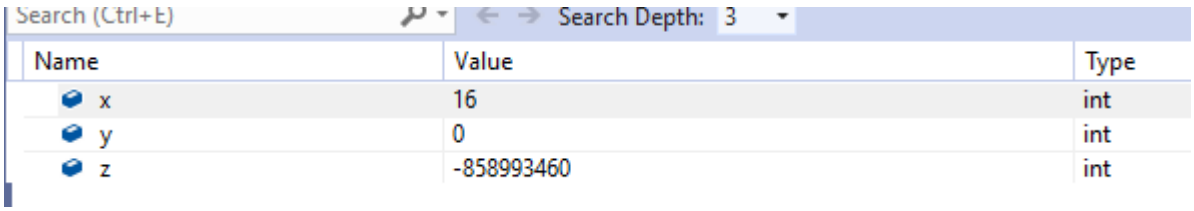


As you have toggled a break point at line# 7 therefore execution of the program stops at this line.

The explorer and console at this point is shown below

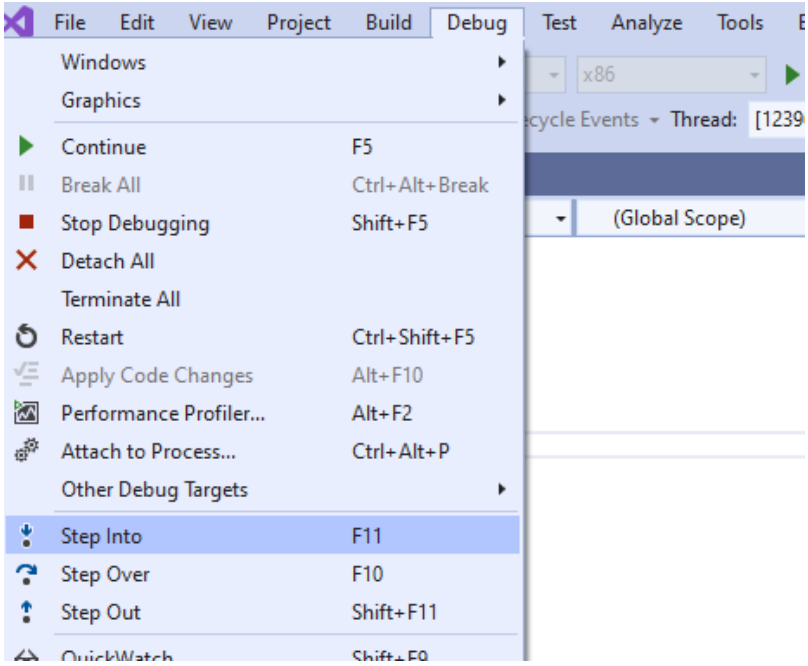


You can notice a small yellow arrow sign at that break point line. At this point, the program is executed till line# 7. Till line# 7, the value of variables are shown as below. Notice that variable z is only declared at line#7 therefore it has some garbage value. Nothing has been shown in console window yet because your code (till line#7) does not has any cout statement.

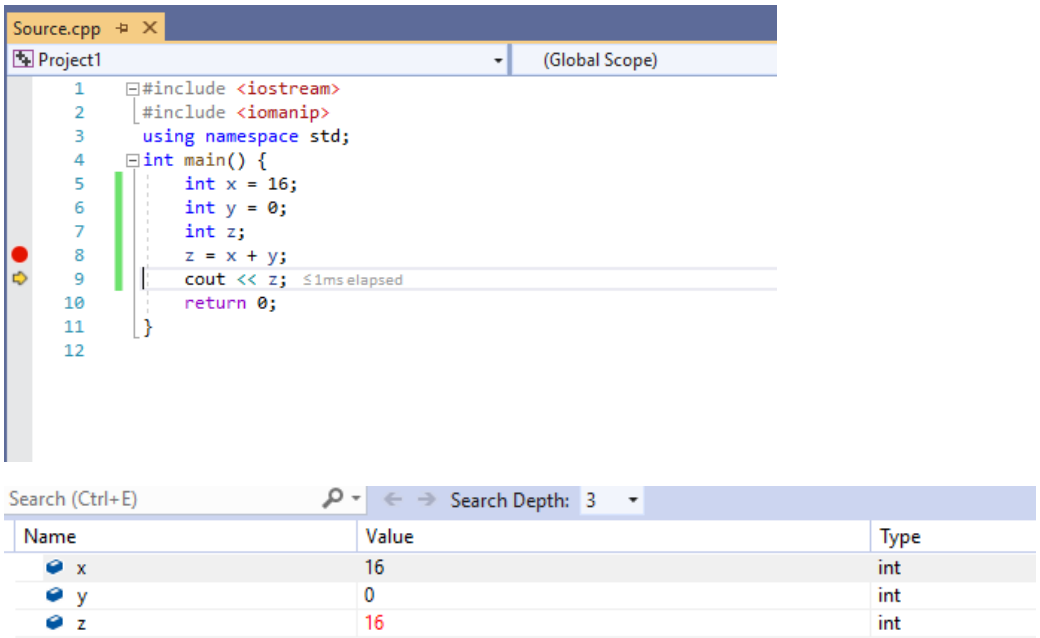


Step 4:

Proceed with the code execution. For that, go to **Debug->Step into** or **press F11**.

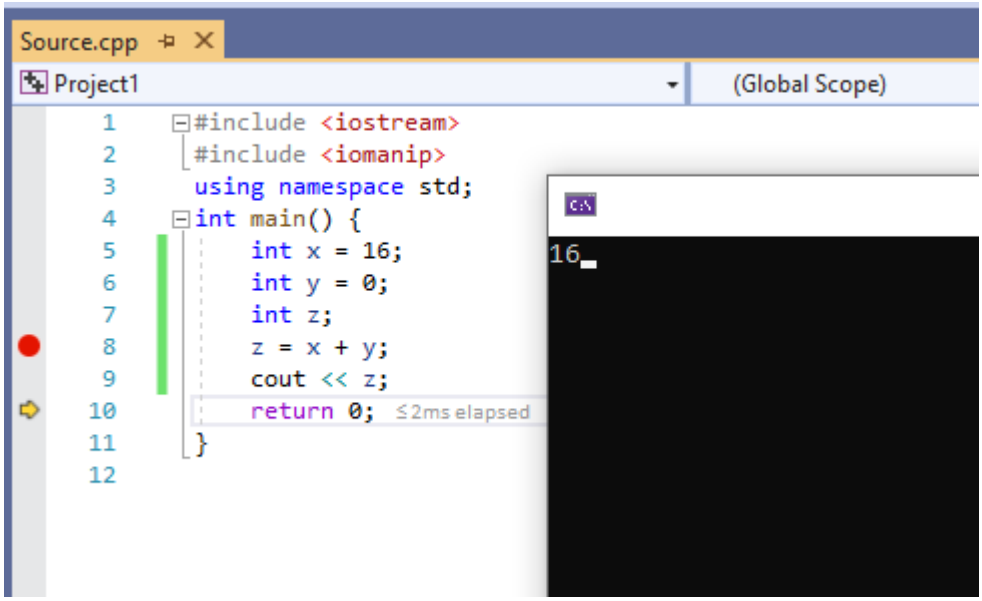


Once you do that following changes are made to the variables. You'll notice that arrow now moves to line#8. Again press F11 to execute this current line. The value of variables will be updated as shown below



Since line#8 updates the value of z, therefore z is updated with the value 16. Console screen is still empty due to same reasons as discussed.

Proceed forward and again press F11. Now, line# executes a cout statement therefore console will reflect this change.



Note: If you choose **Continue** option from the top toolbar then the program does not executes line wise one by one. Instead, it executes in one go (as it does when we run without debugging).

Programming fundamentals

Lecture 7: Switch statement, Nested if, loops

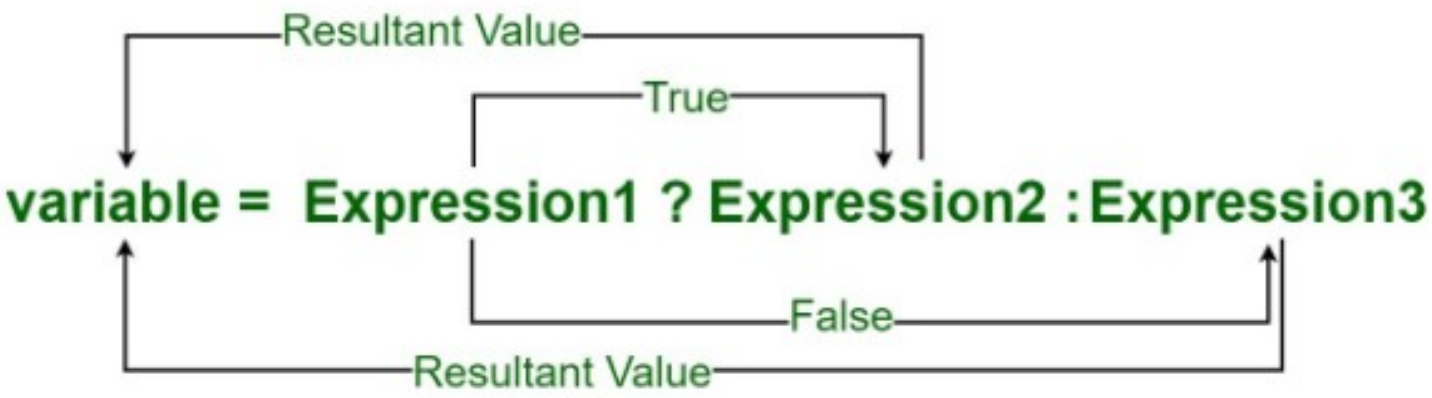
Recap

- operators, and
- conditional statements

Agenda

- Conditional operator
- Nested if
- Switch Statement, Break Statement, goto Statement
- Repetitive structure: while loop, break/continue

Conditional operator



OR

Consider this code:

```
if ( n1 > n2 )
{
  max = n1;
}
else
{
   max = n2;
}
```

OR

```
#include <iostream>
Using namespace std;
int main()
{
// variable declaration
int n1 = 5, n2 = 10, max;

// Largest among n1 and n2
max = (n1 > n2) ? n1 : n2;

// Print the largest number
Cout<<"Largest number between“;
Cout<<n1<< n2<< max<<endl;

return 0;
}
```

```
cout << ( grade >= 60 ? "Passed" : "Failed" );
```

OR

```
grade >= 60 ? cout << "Passed" : cout << "Failed";
```

If we want to print something

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    int n1 = 5, n2 = 10, max;

    // Largest among n1 and n2
    string m = (n1 > n2) ? "n1 is greater" : "n2 is greater";
    cout << m;
return 0;
}
```

Nested if…else *Statements*

- if…else selection statements *inside* other if…else selection statements
- C++ compiler always associates an else with the *immediately preceding*

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x and y are > 5";
   else
      cout << "x is <= 5";
```

```
if ( x > 5 )
{
   if ( y > 5 )
      cout << "x and y are > 5";
}
else
   cout << "x is <= 5";
```

```
int x = 9,y=3;
```

Microsoft Visual Studio Debug Console

```
x is <=5
```

Some built-in functions

- Rand() #include <cstdlib>
- **sqrt( )**, **sin( )**, **cos( )**, **tan( )**, **log( )**, **ceil( )**, and **floor( )** #include<cmath>.
- Generating random number between a range

Example: nested if

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
     int magic; // magic number
     int guess; // user's guess
     magic = rand(); // get a random number
     cout << "Enter your guess: ";
     cin >> guess;
     if (guess == magic) {
          cout << "** Right **\n";
          cout << magic << " is the magic number.\n";
     }
     else {
          cout << "...Sorry, you're wrong.";
          // use a nested if statement
          if (guess > magic)
                    cout << " Your guess is too high.\n";
          else
                    cout << " Your guess is too low.\n";
          }
     return 0;
}
```

Switch statement

- the value of an expression is successively tested against a list of integer or character constants.
- When a match is found, the statement sequence associated with that match is executed. The general form of the **switch** statement is

```
switch(expression) {
        case constant1:
                statement sequence
                break;
        case constant2:
                statement sequence
                break;
        case constant3:
                statement sequence
                break;
                .
                .
                .
        default:
                statement sequence
}
```

- **break** statement causes program flow to exit from the entire **switch** statement and resume at the next statement outside the **switch**
- The **switch** expression must evaluate to either a character or an integer value. (Floating-point expressions, for example, are not allowed.) Frequently, the expression controlling the **switch** is simply a variable.
- The **default** statement sequence is performed if no matches are found. The **default** is optional; if it is not present, no action takes place if all matches fail.
- When a match is found, the statements associated with that **case** are executed until the **break** is encountered or, in the case of the **default** or the last **case**, the end of the **switch** is reached.

Difference from if statements

- The **switch** differs from the **if** in that **switch** can test only for equality (i.e., for matches between the **switch** expression and the **case** constants)

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cin >> num;
    switch (num)
    {
        case 1:
                cout << "You entered 1\n";
                break;
        case 2:
                cout << "You entered 2\n";
                break;
        case 3:
                cout << "You entered 3\n";
                break;
        default:
                cout << "Default case.\n";
    }
    return 0;
}
```

Nested switch

```
switch(ch1)
{
    case 'A’:
        cout << "This A is part of outer switch";
        switch(ch2) {
            case 'A':
                cout << "This A is part of inner switch";
                break;
            case 'B': // ...
        }
        break;
    case 'B': // ...
```

Goto statement

```
#include <iostream>
using namespace std;
int main()
{
     int num;
     cin >> num;
     abc:
     cout << "control shifted after goto"<<endl;
     switch (num) {
          case 1:
               cout << "You entered 1\n";
               break;
          case 2:
               cout << "You entered 2\n";
               break;
          case 3:
               cout << "You entered 3\n";
               goto abc;
                    break;
          default:
                    cout << "Default case.\n";
     }

     return 0;
}
```

Loops

- While
- Do-while
- for

While loop

```
#include <iostream>
using namespace std;
int main()
{
    int x = 0;
    cout << x << endl;
    cout << x+1 << endl;
    cout << x + 2 << endl;
    cout << x + 3 << endl;
    cout << x + 4 << endl;
    cout << x + 5 << endl;
    cout << x + 6 << endl;
    cout << x + 7 << endl;
    cout << x + 8 << endl;
    cout << x + 9 << endl;
    cout << x + 10 << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int x = 0;
    while (x != 11)
    {
        cout << x<<endl;
        x += 1;
    }
    return 0;
}
```

Another example..

```
#include <iostream>
using namespace std;
int main()
{
    int num = 0;
    while (num != 100)
    {
        cout << "Enter a number (100 to stop): ";
        cin >> num;
    }
    return 0;
}
```

Infinite loop

```
#include <iostream>
using namespace std;
int main()
{
    int x = 0;
    while (1)
    {
        cout << x<<endl;
        x += 1;
    }
    return 0;
}
```

continue

- To bypass the loop's normal control structure
- The **continue** statement forces the next iteration of the loop to take place, skipping any code between itself and the conditional expression that controls the loop

```
#include <iostream>
using namespace std;
int main()
{
int x = 0; int num = 0;
while(x <= 5)
{
    x++;
    cout << "before if iteration#" << x << endl;
    if ((num++) % 2==1) continue;
    cout << x << endl;
    cout << "after if iteration#" << x << endl;

}
return 0;
}
```

These statements will be skipped if **continue** gets executed

Microsoft Visual Studio Debug Console

```
before if iteration#1
1
after if iteration#1
before if iteration#2
before if iteration#3
3
after if iteration#3
before if iteration#4
before if iteration#5
5
after if iteration#5
before if iteration#6
```

Using break to Exit Loops

```
#include <iostream>
using namespace std;
int main()
{
int x = 0; int num = 0;
while(x <= 5)
{
    if ((num++) % 2 == 1)
        break;
    cout << "Num: " << num << endl;

}
return 0;
}
```

Recommended reads

- Dietal & Dietal
  - Chapter 4

Bitwise complement or NOT in C++

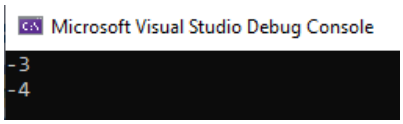
Signed numbers generally follow a 2's complement representation (invert bits and add 1). Thus, when you apply negation, C++ finds 2s complement of a number and return the answer. For now, if you don't know about 2s complement and logic behind it that why we use 2's complement for dealing with signed integers (+ve and -ve) then you can simply ignore. You will hopefully cover that in DLD course. Currently, just note the following rule and don't confuse yourself. I will not give any question in exam/ quiz that asks you to dry run and provide detailed working of bitwise AND,OR,NOR operations and their resultant values.

For any integer n, bitwise complement of n will be -(n+1).

Consider the code:

```
#include <iostream>
using namespace std;
int main()
{
	int a=2,b=3;
	int result_not_a = ~(a);
	int result_not_b = ~(b);
	cout << result_not_a << endl;
	cout << result_not_b << endl;
	return 0;
}
```

Output:



How value of ~a is found?

	…	64	32	16	8	4	2	1
a	0	0	0	0	0	0	1	0

	…	64	32	16	8	4	2	1
b	0	0	0	0	0	0	1	1

After applying NOT

	…	64	32	16	8	4	2	1	In Decimal
a	0	0	0	0	0	0	1	0	2
	1	1	1	1	1	1	0	1	253

After 2s complement of a result (253):

…	64	32	16	8	4	2	1		In decimal
1	1	1	1	1	1	0	1		
0	0	0	0	0	0	1	0	***Invert***	***2s complement***
						+	1	***Add 1***	
0	0	0	0	0	0	1	1		3

Thus, we get an answer -3 for ~a.

Programming fundamentals

Lecture 8: Loops, Nested loops, pattern printing with nested loops

Recap

- Switch statement
- While loop

Agenda

- Do while loop
- For loop
- Nested loops

Typecasting: implicit

```
double d = 3.4;
int i = 3;
int i1 = 0; double d1 = 0;
//double to int
i1 = d;
cout <<"i1: "<< i1 << endl;
//int to double
d1 = i;
cout <<"d1: "<< d1 << endl;
```

Microsoft Visual Studio Debug Console

```
i1: 3
d1: 3
```

Typecasting: explicit

```
double d = 3.4;
int i = 3;
int i1 = 0; double d1 = 0;
i1 = (int)d;
i1 = static_cast<int>(d);
cout <<"i1: "<< i1 << endl;

d1 = (double)i;
cout <<"d1: "<< d1 << endl;
```

Microsoft Visual Studio Debug Console

```
i1: 3
d1: 3
```

DIVISION RULES

```
double d = 3.4;
int a = 5, b = 2;
int res_int = a/b; double res_double = a/b;
cout <<"res_int: "<< res_int << endl;
cout << "res_double: " << res_int << endl;
```

Microsoft Visual Studio Debug Console

```
res_int: 2
res_double: 2
```

```
double d = 3.4;
double a = 5; double b = 2;
int res_int = a/b;
double res_double = a/b;
cout <<"res_int: "<< res_int << endl;
cout << "res_double: " << res_double << endl;
```

Microsoft Visual Studio Debug Console

```
res_int: 2
res_double: 2.5
```

One of the variables must be double to retain the decimal part while dividing

```
double d = 3.4;
int a = 5; double b = 2;
int res_int = a/b;
double res_double = a/b;
cout <<"res_int: "<< res_int << endl;
cout << "res_double: " << res_double << endl;
```

Microsoft Visual Studio Debug Console

```
res_int: 2
res_double: 2.5
```

Some examples..

- cout<< (char)65; **//prints letter A**
- cout<< static_cast<char> (65);

```
for (int x = 65; x < 90; x++)
{
        cout << x << " " << (char)x << "";

}
```

Microsoft Visual Studio Debug Console

```
65 A        66 B        67 C        68 D        69 E        70 F        71 G        72 H
73 I        74 J        75 K        76 L        77 M        78 N        79 O        80 P
81 Q        82 R        83 S        84 T        85 U        86 V        87 W        88 X
89 Y
```

Comparing int with double

```
double d = 3.4;
int i = 3;
if (i == d)//d= 3.3999999999999999   i=3
        cout << "same" << endl;
else
        cout << "different" << endl;
```

Microsoft Visual Studio Debug Console

```
different
```

```
double d = 3.4;
int i = 3;
if (i == (int)d)//d= 3   i=3
        cout << "same" << endl;
else
        cout << "different" << endl;
```

Microsoft Visual Studio Debug Console

```
same
```

```
double d = 3.4;
int i = 3;
if ((double)i == d)//d= 3   i=3
        cout << "same" << endl;
else
        cout << "different" << endl;
```

Microsoft Visual Studio Debug Console

```
different
```

Int to other types

- Int to char

```
int res_int = 65;
char c;
c = static_cast<char>(res_int);
cout <<"c: "<< c << endl;
```

Microsoft Visual Studio Debug Console

```
c: A
```

- Int to string (NOT ALLOWED)

double to other types

- double to char

```
int res_double = 65;
char c;
c = static_cast<char>(res_double);
cout <<"c: "<< c << endl;
```

Microsoft Visual Studio Debug Console

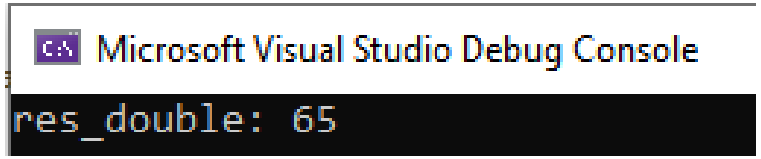
```
c: A
```

- double to string (NOT ALLOWED)

Char to other types

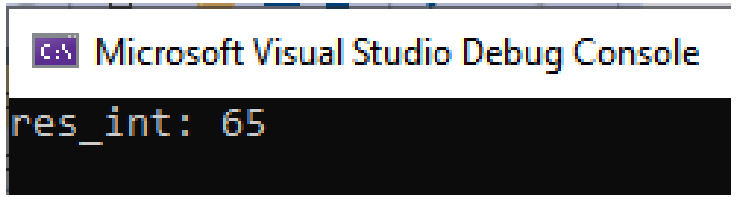
- Char to double

```
double res_double = 65.5;
char c=65;
res_double = static_cast<double>(c);
cout <<"res_double: "<< res_double << endl;
```



- Char to int

```
int res_int = 65.5;
char c=65;
res_int = static_cast<int>(c);
cout <<"res_int: "<< res_int << endl;
```



- char to string (NOT ALLOWED)

Find the number of digits after a decimal part

```
1. double no = 3.44;
2. int count = 0;
3. while (no != ((int)no)){
4.      count++;
5.      no = no * 10;}
6. cout<< count;
```

Line#	no	count	(int)no	no!=(int)no ?
1	3.44			
2	3.44	0		
3	3.44	0	3	**3.44 ! = 3**
4	3.44	1	3	
5	34.4	1	3	
3	34.4	1	34	**34.4 ! = 34**
4	34.4	2	34	
5	344.0	2	34	
3	344.0	2	344	**344.0 ! = 344**
6	Print 2			

Continue in while loop

- To bypass the loop's normal control structure
- The **continue** statement forces the next iteration of the loop to take place, skipping any code between itself and the conditional expression that controls the loop

```
#include <iostream>
using namespace std;
int main()
{
int x = 0; int num = 0;
while(x <= 5)
{
    x++;
    cout << "before if iteration#" << x << endl;
    if ((num++) % 2==1) continue;
    cout << x << endl;
    cout << "after if iteration#" << x << endl;

}
return 0;
}
```

These statements will be skipped if **continue** gets executed

Microsoft Visual Studio Debug Console

```
before if iteration#1
1
after if iteration#1
before if iteration#2
before if iteration#3
3
after if iteration#3
before if iteration#4
before if iteration#5
5
after if iteration#5
before if iteration#6
```

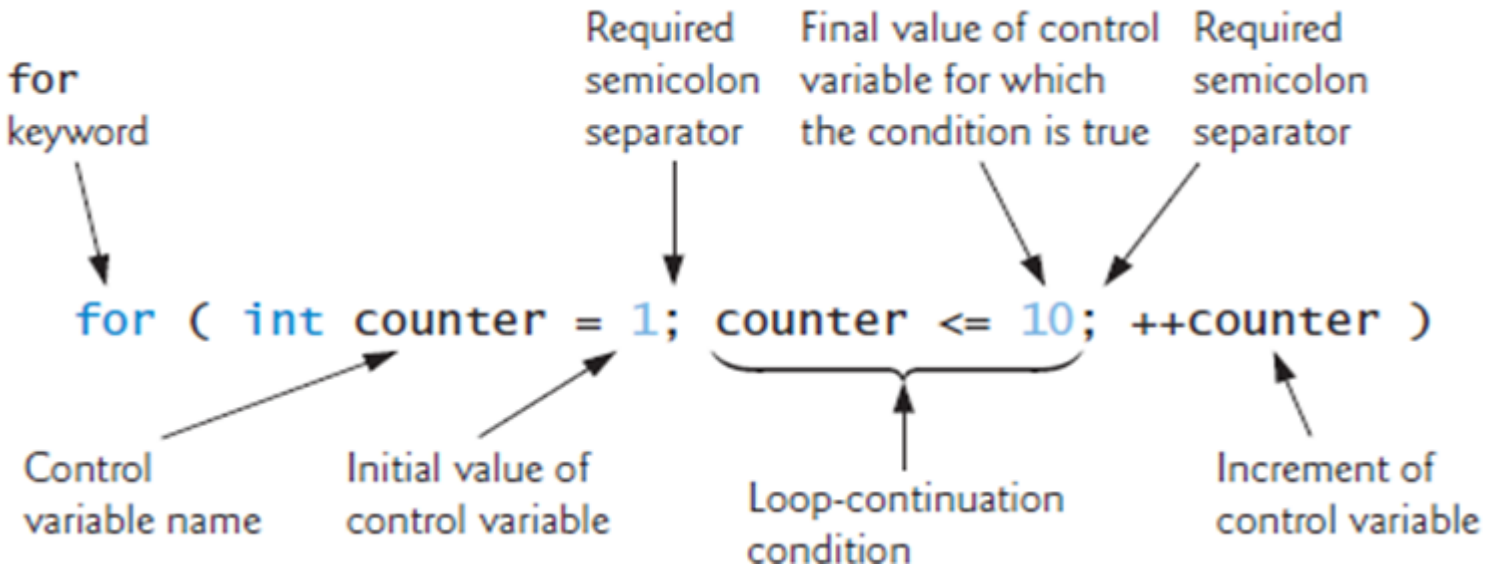
Do while loop

- The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated.
- The syntax of the do...while loop is:

```
do
{
  // statements inside the body of the loop
}
while (testExpression);
```

```
do{
        bake 5 cookies;
} while (thereAreFriends);
```

For loop


for
keyword
Required semicolon separator
Final value of control variable for which the condition is true
Required semicolon separator
for (int counter = 1; counter <= 10; ++counter)
Control variable name
Initial value of control variable
Loop-continuation condition
Increment of control variable

Comparison… print first 10 numbers

```
for (int i=1;i<10;i++)
{
        cout << i << endl;
}
```

```
int i = 1;
while (i<=10)
{
    cout << i << endl;
    i = i + 1;
}
```

```
int i = 1;
do
{
    cout << i << endl;
    i = i + 1;
} while (i <= 10);
```

Nested loops

- Loop within a loop

Pattern practice

```
*****
*   *
*   *
*   *
*****
```

```
@@@@@
$$$$$
&&&&&
```

Division and modulus operator

Number= 12345	
12345 / 10 = 1	12345 % 10 = 5
Number= 2345	
2345 / 10 = 2	2345 % 10 = 5
Number= 6547	
6547 / 10 = 6	6547 % 10 = 7

Extracting each digit of an integer

Number=12345	**d=10000**
Number / d	Number % d
1	2345
Number=2345	**d=1000**
Number / d	Number % d
2	345
Number=345	**d=100**
Number / d	Number % d
3	45
Number=45	**d=10**
Number / d	Number % d
4	5

Palindrome

```
int num=12345;
int d = 10000;
while (num % d!=0)
{
    cout << "Digit: " <<num/d<< endl;
    num = num % d;
    d = d / 10;
}
cout << "Digit: " << num / d << endl;
```

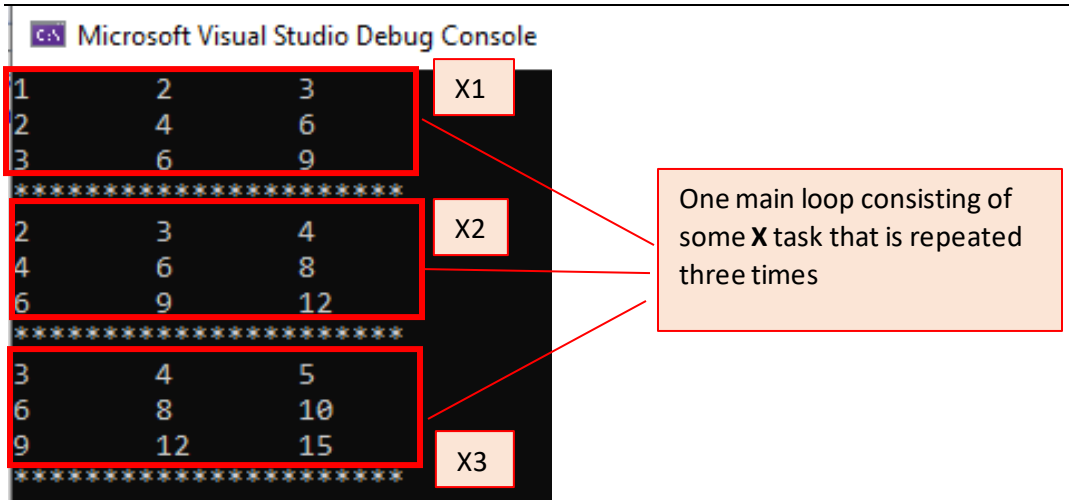
Multiple cases in a switch

```
switch (grade) // switch statement nested in while
{
    case 'A': // grade was uppercase A
    case 'a': // or lowercase a
        ++aCount; // increment aCount
        break; // necessary to exit switch
    case 'B': // grade was uppercase B
    case 'b': // or lowercase b
        ++bCount; // increment bCount
        break; // exit switch
    default: // catch all other characters
        cout << "Incorrect letter grade entered."
        << " Enter a new grade." << endl;
        break; // optional; will exit switch anyway
} // end switch
```

Recommended reads

- Dietal & Dietal
  - Chapter 5

Hint for pattern



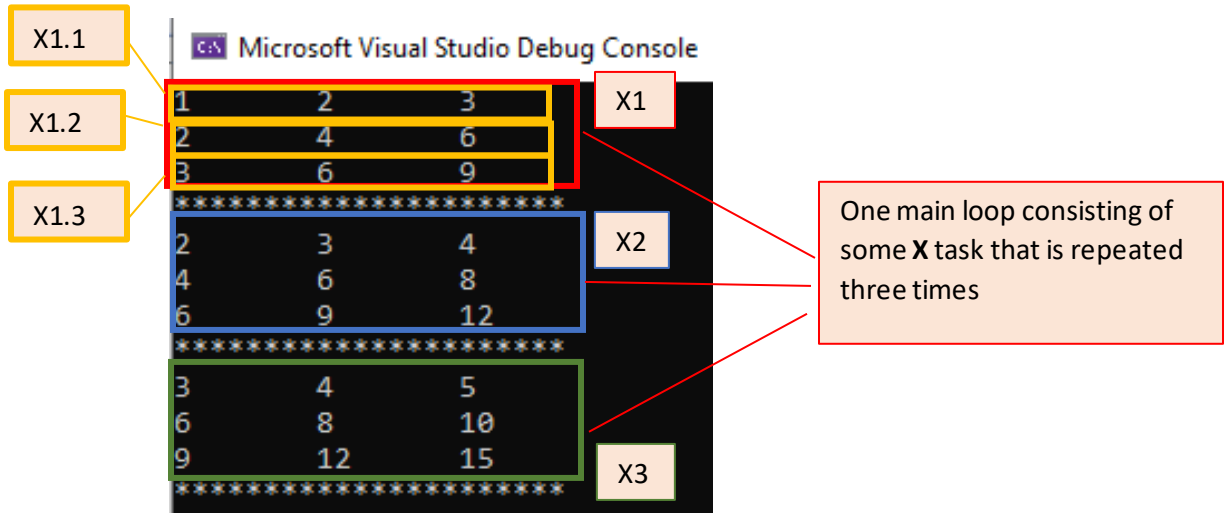
Pattern			Dry run / analysis of pattern		
1	2	3	(1+0) x 1	(2+0) x 1	(3+0) x 1
2	4	6	(1+0) x 2	(2+0) x 2	(3+0) x 2
3	6	9	(1+0) x 3	(2+0) x 3	(3+0) x 3
*********	*********	*********	*********	*********	*********
2	3	4	(1+1) x 1	(2+1) x 1	(3+1) x 1
4	6	8	(1+1) x 2	(2+1) x 2	(3+1) x 2
6	9	12	(1+1) x 3	(2+1) x 3	(3+1) x 3
*********	*********	*********	*********	*********	*********
3	4	5	(1+2) x 1	(2+2) x 1	(3+2) x 1
6	8	10	(1+2) x 2	(2+2) x 2	(3+2) x 2
9	12	15	(1+2) x 3	(2+2) x 3	(3+2) x 3

So you can have one loop that goes from 1 to 3. Thus inside main loop we need to perform the tasks x1, x2 and x3

```
For (i=1 to 3)
{
        //perform the tasks x1
        //perform the tasks x2
        //perform the tasks x3
}
```

Now lets explore the X task (X1 , X2, X3) that has to be repeated 3 times. Each sub task further consists of drawing a 3 by 3 pattern. So we can break down these tasks to smaller sub tasks:

Breaking down X1 into smaller sub tasks:



These subtasks should be nested with out main loop. We can write sub task X1 as:

```
For (i=1 to 3)
{
        //perform the tasks x1
                For (j=1 to 3) Perform x1.1,   x1.2,   x1.3 tasks
                {

                }
        //perform the tasks x2
                For (j=1 to 3) Perform x2.1,   x2.2,   x2.3 tasks
                {

                }
        //perform the tasks x3
                For (j=1 to 3) Perform x3.1,   x3.2,   x3.3 tasks
                {

                }
```

}

If we further sub divide our tasks and analyze the pattern, we can easily see that the three loops above can be replaced with one nested loop (with loop variable as “k”).

```
For (i=1 to 3)
{
        For (j=1 to 3)
        {
                For (k=1 to 3)
                {
                        //insert the logic or formula that can print the required
                        pattern
                }

        }
}
```

Dry run the and execute these loops and see what are the values of loop variables I, j and k. This will help in deciding which values should be added or multiplied together to generate the pattern. You can run the script like the following to see what are the values of loop variable so that you can use them accordingly.

```
#include <iostream>
using namespace std;
int main()
{
        int i, j,k;

        for (i = 1; i <= 5; i++)
        {
                cout << "Value of i:";
                cout<< i << endl;
                cout << "**********************";
                cout << endl;
                for (j = 1; j <= 5; j++)
                {
                        cout << "Value of j:";
                        cout << j << endl;
                        cout << "**********************";
                        cout << endl;
                        for (k = 1; k <= 5; k++)
                        {
                                cout << "Value of k:";
                                cout << k << endl;
                        }
```

```
                cout << "**********************";
                cout << endl;

            }
        }

        return 0;
}
```

Programming fundamentals

Lecture 10: Constant variables and Arrays

Recap

- Loops
- Pattern designing through loops

Agenda

- Constant variables
- Arrays

Recap

- We've already learnt that to store anything we need a memory location
- Then, we explored various types of variables that define the type of data to be stored in memory. E.g. **int variable** indicates that memory location hold by this variable can store the data that has type ***INTEGERS***.
- Just to recap:

```
int sum=43;
```

- The statement means a location on memory is allocated (with some address e.g. **0500**). **sum** is used to identify that location and it holds a value **43**

Constants

- In many scenarios, we have to deal with values that remain fixed throughout our algorithm or program.
- **E.g.** *interest rate in banking applications remain fixed. We may accidently change this variable's value. To avoid that error scenario, in C++ we have* ***constants*** *that deal with this issue.*
- Constants refer to as fixed values, unlike variables whose value can be altered, constants - as the name implies does not change, they remain constant.
- Constant must have to be initialized at the time of creating it, and new values cannot be assigned later to it.
- There are two other different ways to define constants in C++. These are:
  - By using *const* keyword
  - By using *#define* preprocessor

Using const keyword

- Const type const_name;
- It is also possible to put const either before or after the type.

int const SIDE = 50;

or

const int SIDE = 50;

```
#include <iostream>
using namespace std;
int main()
{
    const int SIDE = 50;
    int area;
    area = SIDE*SIDE;
    cout<<"The area of the
    square with side: " << SIDE
    <<" is: " << area << endl;
    return 0;
}
```

#define

- **#define** keyboard is used to define constant by specifying its name and value, separated by spaces:
- #define KILLBONUS 5000
- The constant does not have a type such as int or char.
- The #define directive enables a simple text substitution that replaces every instance of KILLBONUS in the code with 5000.
- The compiler sees only the end result.

```
#include <iostream>
using namespace std;
#define VAL1 20
#define VAL2 6
#define Newline '\n’
int main()
{
    int tot;
    tot = VAL1 * VAL2;
    cout << tot; cout << Newline;
}
```

Extra notes

- Constant variables MUST always be initialized (defined) on declaration time.
- If you try to redefine (re-assign) a constant variable AFTER the code line with declaration, compiler will generate error
- E.g. in the given program, an integer variable named **SIDE** is declared as a constant variable. In second line, SIDE is again assigned value ***43***. As this is not allowed therefore error will be generated.

```
#include<iostream>
using namespace std;
int main()
{
    const int SIDE = 50;
    SIDE = 43;
}
```

(local variable) const int SIDE = 50

Search Online

expression must be a modifiable lvalue

Search Online

Contd..

- Constant variables MUST always be defined during declaration.
- If we skip definition, then error will be generated like shown in the given example.

```
#include<iostream>
using namespace std;
int main()
{
    const int SIDE;
    int area;
    area = SIDE * S
    cout << "The ar
    return 0;
}
```

const variable "SIDE" requires an initializer

Search Online

Arrays

- When we studied variables, we have seen that a variable is allocated one memory cell location when it is defined.
- Variable is just one cell of memory (with some specific address) whereas array is a collection of consecutive memory cells.

Variable A
0001

Cell 1	Cell 2	Cell 3	Cell 4
0010	0011	0012	0013

Memory location

Memory location's address

- The word **consecutive** is important! Array is always consecutive. For example, following is not a valid array!

Cell 1	Cell 2	Cell 3	Cell 4
0010	0051	0012	0013

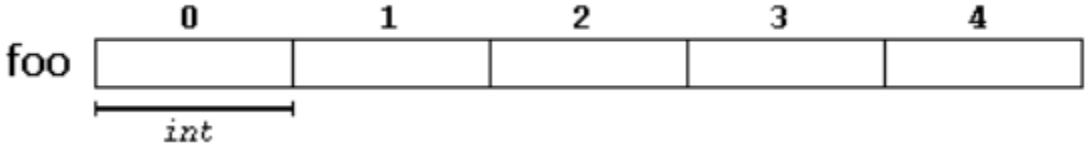
This is because after 0010, the immediate next memory location (0011) must be there as a second cell, **not 0051**

- Another important point is:
  - All the cells in array are of same data type. For example, we cannot have array like the following:

Cell 1	**Cell 2**	**Cell 3**	**Cell 4**
0010	0051	0012	0013
Int type memory location	Int type memory location	double type memory location	char type memory location

Contd..

- An array containing 5 integer values of type int called foo could be represented as:



- The elements field within square brackets [], representing the number of elements in the array, must be a constant expression
- In this case, these are values of type int.
- These elements are numbered from 0 to 4, being 0 the first and 4 the last
- In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.
- Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
type name [elements];        int foo [5];
```

Integer arrays: Declaration

- For now, we will focus on **integer type arrays**. In future, we will see that we can define other type of arrays also i.e. char array, double array.
- To declare an array the basic format is:

Data type variable name [size of array]

e.g. int age[5];

- Just as we name variables, we can give a meaningful name to an array. In this example an array is created that has the capacity of five. In other words, we can store the age of five persons in it.
- **Declare an array:** int age [5];
- **Declare and definition:** int age [5] = { 16, 2, 77, 40, 11 }; To define at the same time when we declare, we need to specify values in comma separated format and put them into curly brackets.

Access elements of array

- You can access elements of an array by using indices.
- Suppose you declared an array mark as above. The first element is mark[0], second element is mark[1] and so on.

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]

Storing data in an array + print an array

- Just as in case of variable, to access the location we use variable name for correspondence.
- In array, to access a particular element of an array we need to specify **array index. Index** is an integer assigned to each cell of array. It starts from 0 and goes to n-1. That is, to access third location we need to write **age[2].** As the index is starting from 0 so index number will always be one less than the location number. (i.e. 3-1=2 in this case.) Last element's index is always n-1.
- Rest of the procedure is the same as we have done with variable. The only difference is that for arrays, we need to access only one element at a time.

Variable vs arrays

```
int age[3];
age[0]=22;
age[1]=20;
age[2]=45;
cout<< age[0];
cout<< age[1];
cout<< age[2];
```

	22	20	45
age	0	1	2

```
int age1, age2, age3;
age1 =22;
age2 =20;
age3 =45;
cout<< age1;
cout<< age2;
cout<< age3;
```

Practice program

- Write a program to take marks of four students as an input and then display their sum. Use array to store marks.

```
#include<iostream>
using namespace std;
int main()
{
    int marks[4]; int sum = 0;
    cout << "Enter the marks of first student: " <<
    endl;
    cin >> marks[0];
    cout << "Enter the marks of second student: " <<
    endl;
    cin >> marks[1];
    cout << "Enter the marks of third student: " <<
    endl;
    cin >> marks[2];
    cout << "Enter the marks of fourth student: " <<
    endl;
    cin >> marks[3];
    sum = marks[0] + marks[1] + marks[2] + marks[3];
    cout << "SUM: " << sum<< endl;
return 0;
}
```

Recommended reads

- Walter Savitch, Problem Solving with C++ The Object of Programming
  - Chapter 7, page 378, section 7.1

Programming fundamentals

Lecture 11: Sorting in arrays, different data type arrays

Recap

- Constants
- Introduction to arrays
- Basic input/output using arrays

Example: Printing the address of array cells

```
#include<iostream>
using namespace std;
int main()
{
    int C[5] = {5,7,9,6,5};
    for (int i = 0; i < 5; i++)
    {
        cout << "Address of "<<i<<": " << &C[i] << endl;
    }

return 0;
}
```

Microsoft Visual Studio Debug Console

```
Address of 0: 007FF890
Address of 1: 007FF894
Address of 2: 007FF898
Address of 3: 007FF89C
Address of 4: 007FF8A0
```

Example: Base address

- Base address refers to the address of first cell of an array i.e. the cell at index 0 (arr[0]).
- Name of an array acts as a base address. That is, if you print it you get base address displayed

```
#include<iostream>
using namespace std;
int main()
{
    int C[5] = {5,7,9,6,5};
    //base address:
    cout << C << endl;
    for (int i = 0; i < 5; i++)
    {
        cout << "Address of "<<i<<": " << &C[i] << endl;
    }
return 0;
}
```

Microsoft Visual Studio Debug Console

```
0055FB38
Address of 0: 0055FB38
Address of 1: 0055FB3C
Address of 2: 0055FB40
Address of 3: 0055FB44
Address of 4: 0055FB48
```

Example: Postfix and prefix increment with arrays

- Rules of prefix/postfix are applied accordingly on array's index
- What if an index of array and variable on which increment operator is applied are the same?

Postfix

```
#include<iostream>
using namespace std;
int main()
{
    int C[5] = {5,7,9,6,5};int j = 0;
    cout << "Postfix example" << endl;
    cout << "contents before:" << endl;
    for (int i = 0; i < 5; i++)
        cout << C[i] << " ";
    cout << endl;

    C[j] = j++;//postfix increment, index
    is taken as older j value
    cout << "contents after:" << endl;
    for (int i = 0; i < 5; i++)
        cout << C[i] << " ";
    cout << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Postfix example
contents before:
5 7 9 6 5
contents after:
0 7 9 6 5
```

Prefix

```
#include<iostream>
using namespace std;
int main()
{
    int C[5] = {5,7,9,6,5};int j = 0;
    cout << "Prefix example" << endl;
    cout << "contents before:" << endl;
    for (int i = 0; i < 5; i++)
        cout << C[i] << " ";
    cout << endl;

    C[j] = ++j;//prefix increment, index is
    taken as a new updated j value
    cout << "contents after:" << endl;
    for (int i = 0; i < 5; i++)
        cout << C[i] << " ";
    cout << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Prefix example
contents before:
5 7 9 6 5
contents after:
5 1 9 6 5
```

Agenda

- Unsized array initialization
- Searching an array
- Array with different data types
- Sorting
- Char array as a string
- Multi dimensional arrays

Unsized array initialization

```
#include<iostream>
using namespace std;
int main()
{
    int C[] = { 5,7,9,6,5 };
    for (int i = 0; i < 5; i++)
    {
        cout << C[i] << " ";
    }

    return 0;
}
```

Searching an array

```
#include<iostream>
using namespace std;
int main()
    {
    int C[5] = { 5,7,9,6,5 };int element = 5;
    for (int i = 0; i < 5; i++)
    {
        if (C[i] == element)
        {
            cout << "Element found at index " << i;
            break;
        }
    }

    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Element found at index 0
```

Array of various data types

- There can be different types of arrays
- Note: you cannot have different data type arrays but with the same name!

```
#include<iostream>
using namespace std;
int main()
{
    double A[] = { 2.4,5.6,7.8 };
    char B[] = { 'a','b','c' };
    bool C[] = { true,false,false };
    int D[] = { 3,5,6 };
    string E[] = { "amna", "usama"};
    return 0;
}
```

Sorting

- sorting a list of values, such as a list of sales figures that must be sorted from lowest to highest or from highest to lowest, or a list of words that must be sorted into alphabetical order.

	Sales[0]	Sales[1]	Sales[2]	Sales[3]	Sales[4]
Sales	1	2	3	4	5
Index	0	1	2	3	4

- An array is sorted only if
  - Sales[0] <= Sales[1] <= Sales[2] <= Sales[3] <= Sales[4]

Check if array is sorted

```
#include<iostream>
using namespace std;
int main()
{
    int D[] = { 1,2,3,4,5 };
    for (int i = 0; i < 4; i++)
    {
        if (D[i] > D[i + 1])
        {
            cout << "The array is
            not sorted" << endl;
            break;
        }
    }
    return 0;
}
```

Compare…

i=0

1	2	3	4	5
0	1	2	3	4

i=1

1	2	3	4	5
0	1	2	3	4

i=2

1	2	3	4	5
0	1	2	3	4

i=3

1	2	3	4	5
0	1	2	3	4

Swap two numbers

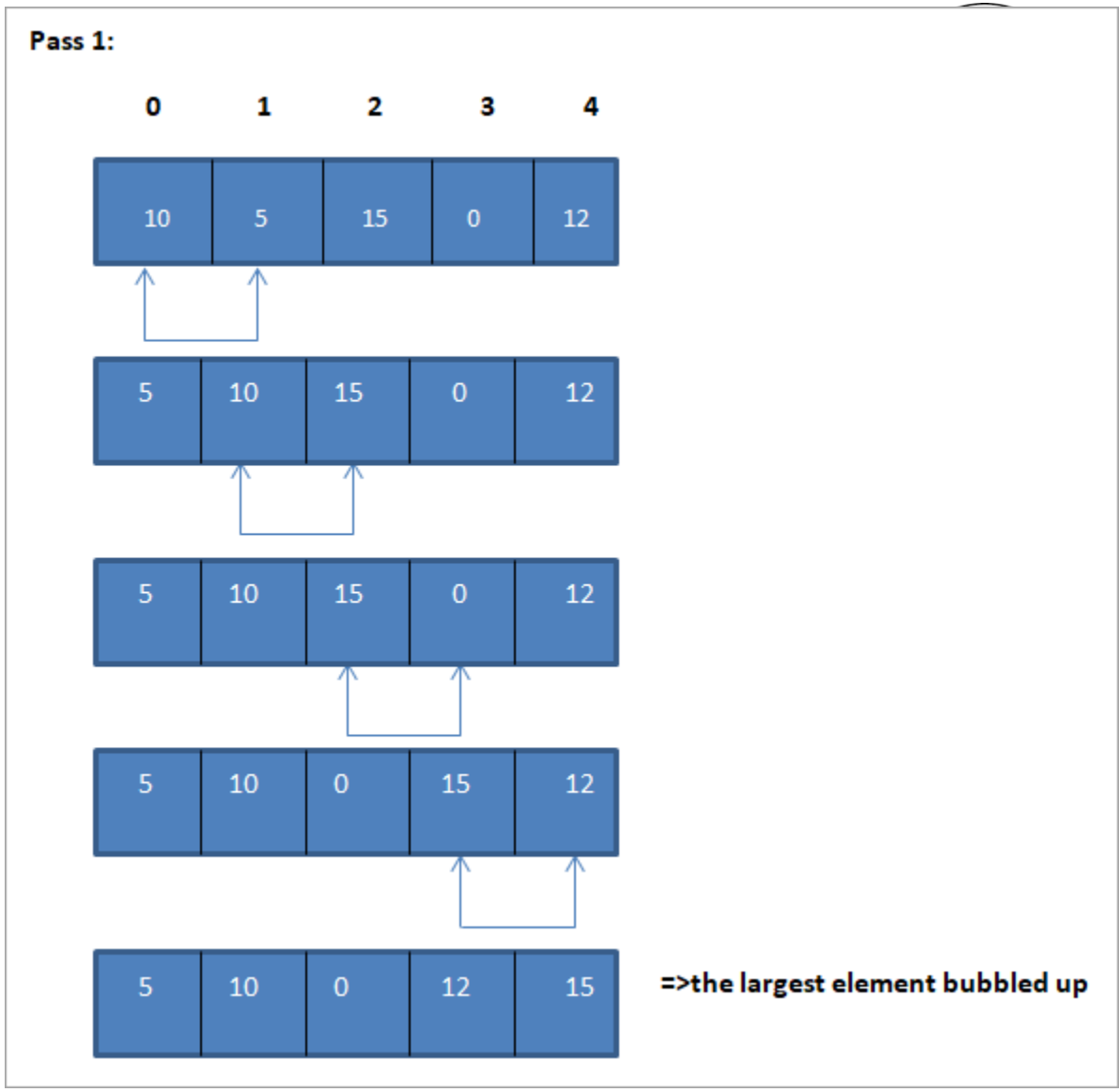
```
#include<iostream>
using namespace std;
int main()
{
    int a,b,temp;
    a = 4;
    b = 5;
    cout << "a:" << a << endl;
    cout << "b:" << b << endl;
    temp = a;
    a = b;
    b = temp;
    cout << "a:" << a << endl;
    cout << "b:" <<b<< endl;
    return 0;
}
```

How to sort?

- There are many sorting algorithms, insertion sort, bubble sort, selection sort, etc.
- We will discuss only the simple algorithm (bubble sort) as other algorithms are not in our scope.

Bubble sort
(Optional to study for practice)

- Works by repeatedly swapping the adjacent elements if they are in wrong order
- Multiple passes to an array
- In each pass, iterate and compare consecutive element pairs..e.g.
  - Index 0 and index 1
  - Index 1 and index 2
  - Index 2 and index 3
  - Index 3 and index 4
- After each pass, if the comparison is done on the basis of > operator the largest element is shifted to the end of an array



Pass 2:

0	1	2	3	4
5	10	0	12	15
5	10	0	12	15
5	0	10	12	15
5	0	10	12	15
5	0	10	12	15

=>second largest element bubbled

Pass 3:

5	0	10	12	15
0	5	10	12	15
0	5	10	12	15
0	5	10	12	15
0	5	10	12	15

=>third element bubbled up, list sorted

```
#include<iostream>
using namespace std;
int main()
{
int n, i, arr[50], j, temp;
cout << "Enter total number of elements :";
cin >> n;
cout << "Enter " << n << " numbers :";
for (i = 0; i < n; i++)
        cin >> arr[i];
cout << "Sorting array using bubble sort technique...\n";
for (i = 0; i < (n - 1); i++)
{
    for (j = 0; j < (n - i - 1); j++)
    {
        if (arr[j] > arr[j + 1])
        {
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}
return 0;
}
```

```
cout << "Sorted list in ascending order :\n";
for (i = 0; i < n; i++)
{
cout << arr[i] << " ";
}
return 0;
}
```

Reading

- Walter savitch, problem solving with C++
  - Pg 421, Sorting an array

Strings or char array

H	e	l	l	o	'\0'

- In C++, a *string is* defined as a character array that is terminated by a null.
- A null character is specified using **'\0'**, and is zero.
- Because of the null terminator, it is necessary to declare a character array to be one character longer than the largest string that it will hold.

```
char str[11];
```

- Specifying the size as 11 makes room for the null at the end of the string.

```
char str[6] = "hello";
```

This is the same as writing

```
char str[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

```
#include<iostream>
using namespace std;
int main()
{
char a[5] = {'h','e','l','l','o'};
char a[5] = { "hello" };
return 0;
}
```

```
int main()
{
    char a[5] = {'h','e','l','l','o'};
    char a[5] = { "hello" };

    return 0;
}
```

a value of type "const char [6]" cannot be used to initialize an entity of type "char [5]"

Search Online

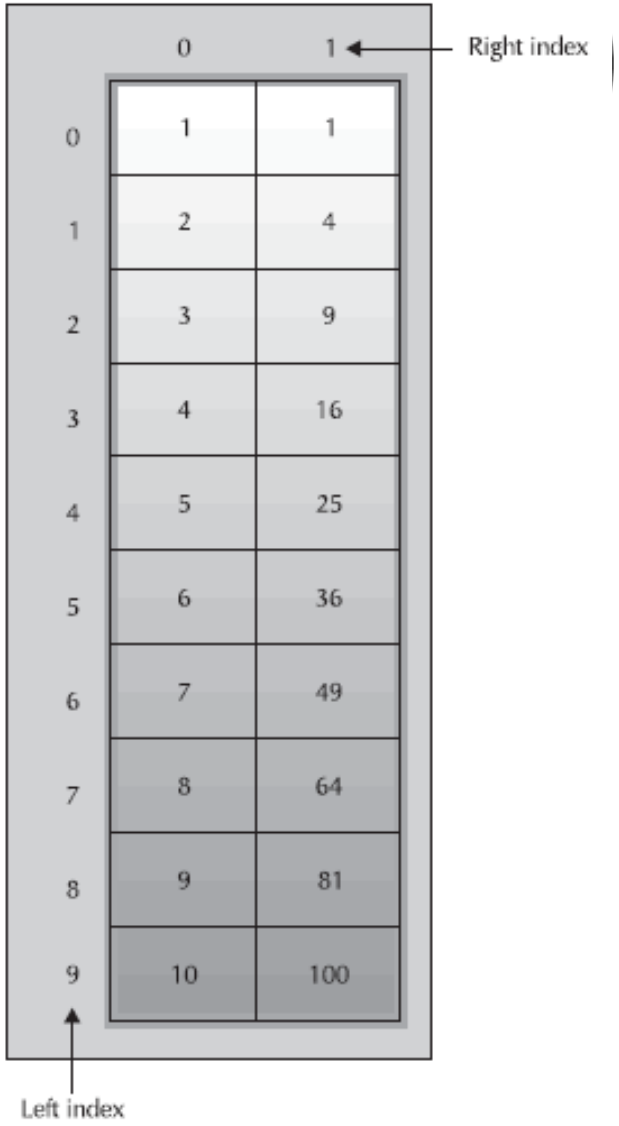
```
#include<iostream>
using namespace std;
int main()
{
    char a1[5] = {'h','e','l','l','o'};
    char a[6];
    cin >> a;
    for(int i=0;i<7;i++)
    cout << "index "<<i<<" "<<a[i]<<endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
hello
index 0 h
index 1 e
index 2 l
index 3 l
index 4 o
index 5
index 6 ╠
```

Multi-dimensional array

```
#include<iostream>
using namespace std;
int main()
{
    int sqrs1[10][2] = {1, 1,2, 4,3, 9,4, 16,5,
    25,6, 36,7, 49,8, 64,9, 81,10, 100};
    int sqrs2[10][2] = {{1, 1},{2, 4},{3,9},
    {4, 16},{5, 25},{6, 36},{7, 49},{8, 64},{9,
    81},{10, 100}};
    return 0;
}
```



Example

	0	1
0	**1**	**2**
1	**3**	**4**

[0][0]	**[0][1]**
[1][0]	**[1][1]**

```
#include<iostream>
using namespace std;
int main()
{
    const int rows = 2; const int cols = 2;
    int arr[rows][cols] = {1,2,3,4};
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < rows; j++)
        {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Reading

- Walter savitch, problem solving with C++
  - Pg 425, multi-dimensional array basics

Recommended reads

- Walter Savitch, Problem Solving with C++ The Object of Programming
  - Chapter 7, page 378, section 7.1

Programming fundamentals

Lecture 12: 2D arrays, char arrays and strings

Recap

- Basic idea of multi-dimensional arrays
- Finding element in an array
- Sorting
- Working of cin

Agenda

- 2D arrays
- Char array
- Strings

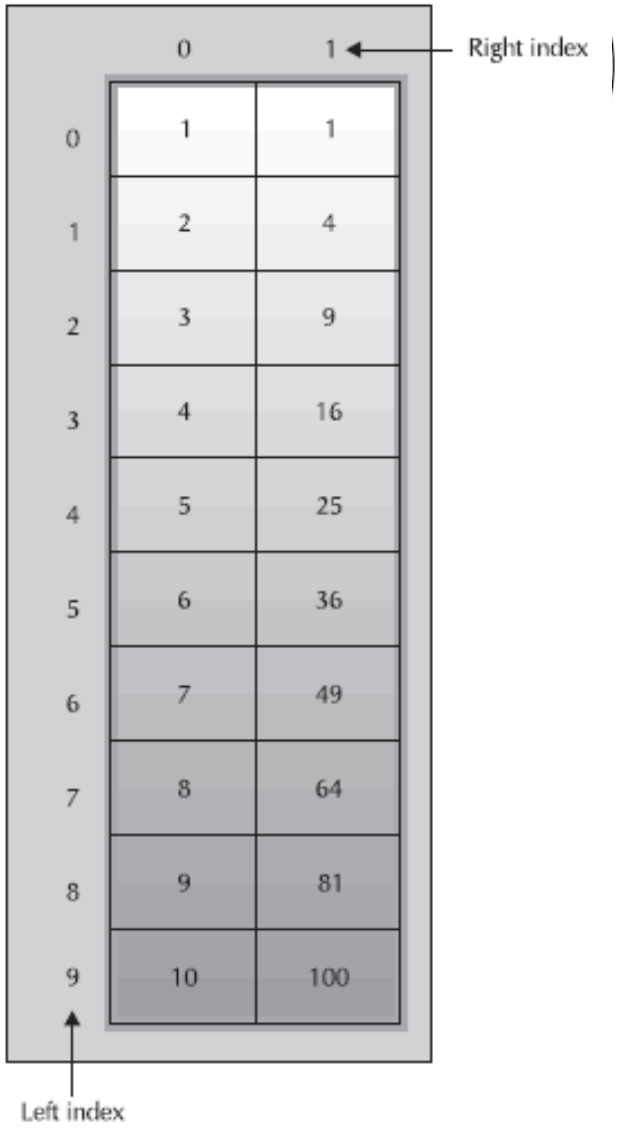
Copying one array **A** to another array **B**

```
#include<iostream>
using namespace std;
int main()
{
    int A[5],B[5];
    for (int i = 0; i < 5; i++)
    {
        cin[ >> A[i];
    }
    for (int i = 0; i < 5; i++)
    {
        B[i]=A[i];
    }
    for (int i = 0; i < 5; i++)
    {
        cout << "A: " << A[i] << " ";
        cout << "B: " << B[i];
        cout << endl;
    }
    return 0;
}
```

2D array

```
#include<iostream>
using namespace std;
int main()
{
    int sqrs1[10][2] = {1, 1,2, 4,3, 9,4, 16,5,
    25,6, 36,7, 49,8, 64,9, 81,10, 100};

    int sqrs2[10][2] = {{1, 1},{2, 4},{3,9},
    {4, 16},{5, 25},{6, 36},{7, 49},{8, 64},{9,
    81},{10, 100}};
    return 0;
}
```

Two Ways to initialize 2D arrays



Example

	0	1
0	**1**	**2**
1	**3**	**4**

[0][0]	**[0][1]**
[1][0]	**[1][1]**

```
#include<iostream>
using namespace std;
int main()
{
    const int rows = 2; const int cols = 2;
    int arr[rows][cols] = {1,2,3,4};
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

	0	1	2	3
0	**1**	**2**	**3**	**4**
1				
2				

```
#include<iostream>
using namespace std;
int main()
{
    const int rows = 3; const int cols = 4;
    int arr[rows][cols];
    for (int i = 0; i < rows; i++)
    {
        if(i==0)
        {
            for (int j = 0; j < cols; j++)
                cin >> arr[i][j];
            cout << endl;
        }
    }
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            cout << arr[i][j] << "\t";
        }
        cout << endl;
    }
return 0;
}
```

String vs char array

- A character array is simply an **array of characters** can terminated by a null character.
- A string is a **class which defines objects** that be represented as stream of characters.
- Size of the character array has to **allocated statically**, more memory cannot be allocated at run time if required. Unused allocated **memory is wasted** in case of character array.
- In case of strings, memory is **allocated dynamically**. More memory can be allocated at run time on demand. As no memory is preallocated, **no memory is wasted**.
- Implementation of **character array is faster** than std:: string. **Strings are slower** when compared to implementation than character array.
- Character array **do not offer** much **inbuilt functions** to manipulate strings. String class defines **a number of functionalities** which allow manifold operations on strings.

char array

```
#include<iostream>
using namespace std;
int main()
{
   char a1[5];
   for (int i = 0; i < 5; i++)
      cin>>a1[i];

   for (int i = 0; i < 5; i++)
      cout << "index " << i << " " << a1[i] << endl;
   return 0;
}
```

Char array can also act as a string

- In C++, a *string is* defined as a character array that is terminated by a null.
- A null character is specified using **'\0'**.
- Because of the null terminator, it is necessary to declare a character array to be one character longer than the largest string that it will hold.

```
char str[11];
```

- Specifying the size as 11 makes room for the null at the end of the string.

- Note: the string data type that we briefly covered is not the same as this character array. When we write "string name;" we are actually creating an object of string class built-in and defined in standard c++ library.
- We will cover string class in a separate lecture. So, ignore it for now.

Strings or char array

H	e	l	l	o	'\0'
0	**1**	**2**	**3**	**4**	**5**

```
char str[6] = "hello";
```

This is the same as writing

```
char str[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

If we use character array as a string and initialize it with a string, then we must consider a space for '\0' character. Otherwise, compiler will generate an error. Like in this example, hello is of 5 characters so you need to declare a char array of 6 size. One extra cell for '\0' character.

```
#include<iostream>
using namespace std;
int main()
{
    char str[5] = "hello";

    return 0;
}
```

a value of type "const char [6]" cannot be used to initialize an entity of type "char [5]"

Search Online

Alternative ways of defining a string

char str[4] = "C++";

char str[] = {'C','+','+','\0'};

char str[4] = {'C','+','+','\0'};

For initialization using curly brackets, we cannot put string inside braces, this is not allowed. For example, see the code below.

```
#include<iostream>
using namespace std;
int main()
{
    char a[5] =
    {'h','e','l','l','o'};
    char a[5] = { "hello" };
    return 0;
}
```

```
int main()
{
    char a[5] = {'h','e','l','l','o'};
    char a[5] = { "hello" };

    return 0;
}
```

a value of type "const char [6]" cannot be used to initialize an entity of type "char [5]"

Search Online

```
#include<iostream>
using namespace std;
int main()
{
    char str[5];
    cin >> str;
    cout << str;
    return 0;
}
```

If we want to use character array as a string, then there is no need to append square brackets with the name of array.

Note: for int, double, bool, and other data types, array name depicts base address. But in case of char array, the array name does not act as a base address.

Like in this example, we have declared an array of 5 characters. Thus, it is mandatory for user to enter only 4 characters (4 plus 1 for '\o'). If the user enters "hello" which is 5 characters long then the compiler will throw exception.

```
#include<iostream>
using namespace std;
int main()
{
    char str[5];
    cin >> str;

    int i = 0;
    while (1)
    {
        if (str[i++] == '\0')
            break;
    }
    return 0;
}
```

We can apply conditional logic on null terminator character to see if we reach the end of our character array or not. This will might help in designing a logic of “finding the number of characters in a string”

In this program, an infinite loop is created that keeps on accessing elements of an array one by one using statement `str[i++]`. We want to end the loop, once end of a character array is reached. Therefore we introduced if statement in it.

Giving input to char array *in string form*

```
#include<iostream>
using namespace std;
int main()
{
    char a[6];
    cin >> a;
    for(int i=0;i<7;i++)
    cout << "index "<<i<<" "<<a[i]<<endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
hello
index 0 h
index 1 e
index 2 l
index 3 l
index 4 o
index 5
index 6
```

Giving input to char array *in characters form*

```
#include<iostream>
using namespace std;
int main()
{
char a[6];
cin >> a;
for (int i = 0; i < 7; i++)
cout << "index " << i << " " << a[i] << endl;
return 0;
}
```

Once you enter a space after character 'h', cin will assume that you have finished giving input. It thus stores 'h' to array. Any other characters that you write after that will be discarded. You will notice that by default the compiler places blanks and special characters into the empty places

Microsoft Visual Studio Debug Console

```
h e l l o
index 0 h
index 1
index 2
index 3
index 4
index 5
index 6
```

What if we want to have a sentence recorded in our char array

Char array takes a space as a terminating character

- "Hello World" is taken as "Hello"… space is taken as terminating character

```
#include<iostream>
using namespace std;
int main()
{
    char str[20];
    cout << "\nEnter another string: ";
    cin >> str;
    cout << "You entered: " << str <<
    endl;
    return 0;
}
```

'world' is discarded because cin has taken space as a terminating character.

Microsoft Visual Studio Debug Console

```
Enter another string: hello world
You entered: hello
```

Solution to get a complete sentence as input in character array

- Use getline function
- Getline takes three arguments: 1) char array name, 2) number of characters to take from an input, 3) terminating character. Argument 3 is optional and if we don't specify it then by default '\n' newline character is assumed.
- Now with getline, cin will either terminate when ***number of characters*** is reached, or when newline (ENTER) character is encountered.

```
#include<iostream>
using namespace std;
int main()
{
    char str[20];
    cout << "\nEnter another string: ";
    cin.getline(str,20);
    cout << "You entered: " << str << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Enter another string: hello World
You entered: hello World
```

Contd..

Cin integer and then cin char array will cause ambiguous behavior

- After you do cin>>t;, the buffer still contains the newline sequence. Then getline() reads an immediate newline which tricks it into thinking that the user just pressed enter without typing anything.
- In order to fix this, you need to ignore the newline before calling getline().

```
#include<iostream>
using namespace std;
int main()
{
    int x;
    char str[6];
    cin >> x;
    cin.getline(str, 6,'\n');
    cout << "You entered: " << str << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
4
You entered:
```

The program terminated without taking input in char array

Contd..

```
#include<iostream>
using namespace std;
int main()
{
    int x;
    char str[6];
    cin >> x;
    cin.ignore();
    cin.get(str, 6,'\n');
    cout << "You entered: " << str << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

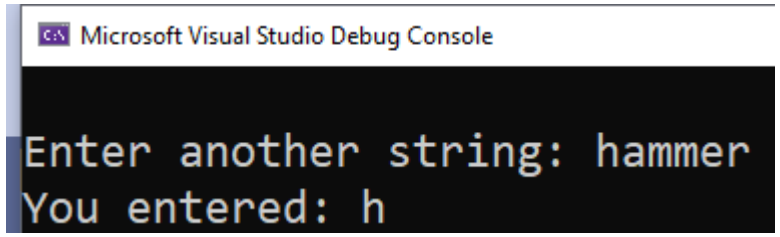
```
4
hello
You entered: hello
```

Contd..

- It also allows a third argument that specifies the terminating character for input. The default value is the newline character.

```
#include<iostream>
using namespace std;
int main()
{
    char str[20];
    cout << "\nEnter another string: ";
    cin.getline(str, 20,'a');
    cout << "You entered: " << str << endl;
    return 0;
}
```

Since ‘a’ is taken as a terminating character. Thus, once the user enters a, cin will stop taking further input. Like ‘a’ inserted after first ‘h’ in below example.



Practice problems

character array

Changing a character from upper case to lower case

ASCII value of A is 65 and ASCII value of a is 97

65+32 = 97 = a = equivalent value of A in lowercase i.e., a

Decimal	Character	Decimal	Character
65	A	97	a
66	B	98	b
67	C	99	c
68	D	100	d
69	E	101	e
70	F	102	f
71	G	103	g
72	H	104	h
73	I	105	i
74	J	106	j

```
#include<iostream>
using namespace std;
int main()
{
    char ch;
    cout << "Enter a character in uppercase :
    ";
    cin >> ch;
    ch = ch + 32;
    cout << "Character in lowercase = " << ch;
    return 0;
}
```

Finding length of a string

```
#include<iostream>
using namespace std;
int main()
{
    char str[100]; /* Declares a string of size 100 */
    int l = 0;

    cin >> str;
    while (str[l] != '\0')
    {
    l++;
    }
    cout << "Length of the string is " << l;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Usama
Length of the string is 5
```

Reverse a string

Length: 5

U	S	A	M	A	\0
0	1	2	3	4	5

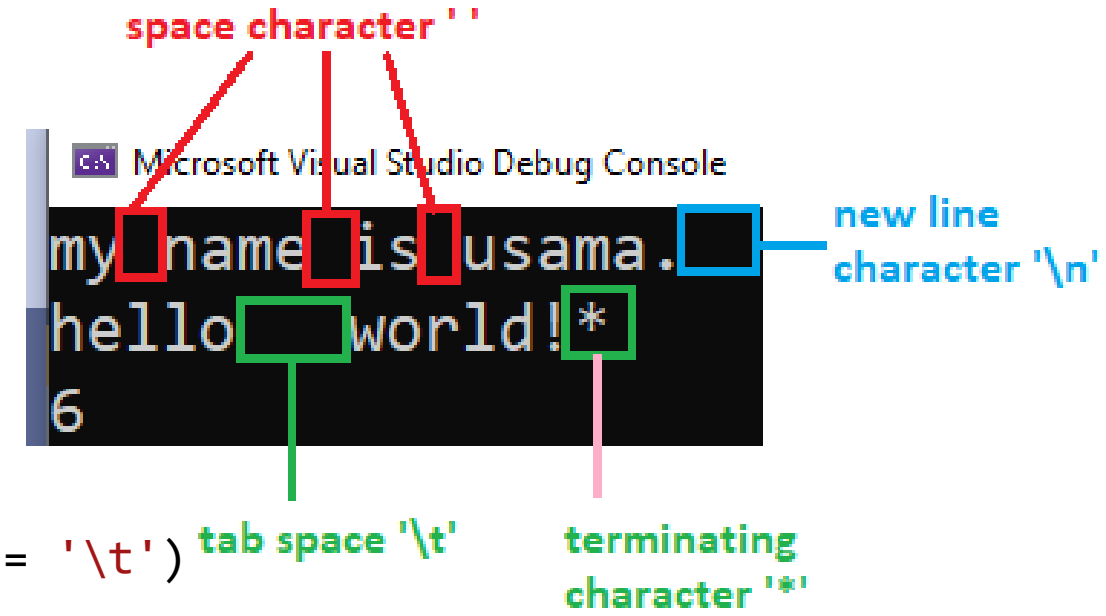
```
#include<iostream>
using namespace std;
int main()
{
    char str[100];
    int l = 0;
    cin >> str;
    while (str[l] != '\0')
    {
        l++;
    }
    for (int i = l-1; i >= 0; i--)
    {
        cout<<str[i];
    }
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
usama
amasu
```

Count the total number of words in a string.

```
#include<iostream>
using namespace std;
int main()
{
    char str[50];
    int i, wrd;
    cin.getline(str,50,'*');//considering * as a
    terminating character
    i = 0;
    wrd = 1;
    while (str[i] != '\0')
    {
    if (str[i] == ' ' || str[i] == '\n' || str[i] == '\t')
    wrd++;
    i++;
    }
    cout<< wrd;
    return 0;
}
```




```
my name is usama.
hello   world!*
6
```

Compare two strings with respect to length

```
#include<iostream>
using namespace std;
int main()
{
char str1[10], str2[10]; /* Declares a string of size
100 */
int l1 = 0, l2 = 0; int i;
cin >> str1;
cin >> str2;
i = 0;
while (str1[i] != '\0')
{
        l1++;i++;
}
cout << "Length of first string is: " << l1<<endl;
i = 0;
while (str2[i] != '\0')
{
        l2++; i++;
}
cout << "Length of second string is: " << l2<<endl;
```

Microsoft Visual Studio Debug Console

```
natalia
zara
Length of first string is: 7
Length of second string is: 4
First string is larger in length than the second string
```

```
if (l1 == l2)
        cout << "Both strings are equal";
else if (l1 > l2)
        cout << "First string is larger in length
        than the second string";
else
        cout << "First string is smaller in length
        than the second string";

return 0;
}
```

Extract a substring

Microsoft Visual Studio Debug Console

```
natalia
Input the position to start extraction :
4
alia
```

```
#include<iostream>
using namespace std;
int main()
{
    char str[10], sstr[10];
    int pos, c = 0; int i; int l = 0;
    cin >> str;
    i = 0;
    while (str[i] != '\0')
    {
        l++; i++;
    }
    cout<<"Input the position to start extraction :"<<endl;
    cin>>pos;
    while (c < l)
    {
        sstr[c] = str[pos- 1+c];
        c++;
    }
    sstr[c] = '\0';
    cout << sstr;
    return 0;
}
```

Some more problems..

- **Copy one string into another string**
- **Count total number of vowels**
- **Extract a substring with starting position as *x* and ending position as *y***
- **Change the letters in input string from upper case to lower case**

Practice problems

String

String Concatenation

+ operator helps in appending a character or a string

```
#include <string>
#include<iostream>
using namespace std;
int main()
{
   string firstName = "Natalia ";
   string lastName = "Chaudhry";
   string fullName = firstName + lastName;
   cout << fullName;
return 0;
}
```

String Concatenation

Use append() function

```
#include <string>
#include<iostream>
using namespace std;
int main()
{
   string firstName = "Natalia ";
   string lastName = "Chaudhry";
   string fullName = firstName.append(lastName);
   cout << fullName;
   return 0;
}
```

+ operator acts according to the situation

- + is used for both addition and concatenation

z will be 30

```
#include <string>
#include<iostream>
using namespace std;
int main()
{
    int x = 10;
    int y = 20;
    int z = x + y;
    cout << z;
    return 0;
}
```

z will be '1020'

```
#include <string>
#include<iostream>
using namespace std;
int main()
{
    string x = 10;
    string y = 20;
    string z = x + y;
    cout << z;
    return 0;
}
```

Find string length: use `length()` function

```
#include <string>
#include<iostream>
using namespace std;
int main()
{
   string txt = "Natalia";
   cout << "The length string is: " << txt.length();
   return 0;
}
```

Microsoft Visual Studio Debug Console

```
The length string is: 7
```

Access Strings

- You can access the characters in a string by referring to its index number inside square brackets []

```
string myString = "Hello";
cout << myString[0];
// Outputs H
myString[0] = ’h';
```

Input in strings

- cin considers a space (whitespace, tabs, etc) as a terminating character, which means that it can only display a single word (even if you type many words)
- Cin.getline() function only works for char array as its first argument
- Use another version of getline():
  - Getline(first argument, second argument);
  - First argument is **cin**
  - Second argument is **string** variable name

Input in strings

```
#include <string>
#include<iostream>
using namespace std;
int main()
{
   string fullName;
   cout << "Type your full name: ";
   getline(cin, fullName);
   cout << "Your name is: " << fullName;
      return 0;
}
```

Microsoft Visual Studio Debug Console

```
Type your full name: Natalia Chaudhry
Your name is: Natalia Chaudhry
```

Reading

- D.S Malik C++ Programming-From Problem Analysis to Program Design, Pg. 486 - Pg. 526, Chapter 9 Array and Strings
- Walter savitch, problem solving with C++
  - Pg 425, multi-dimensional array basics

Recommended reads

- Walter Savitch, Problem Solving with C++ The Object of Programming
  - Chapter 7, page 378, section 7.1

Programming fundamentals

Lecture 13a: Practice questions on *char arrays and strings*

Recap

- Char array
- Strings

Agenda

- Practice questions related to char array and strings

String vs char array

- A character array is simply an **array of characters** can terminated by a null character.
- A string is a **class which defines objects** that be represented as stream of characters.
- Size of the character array has to **allocated statically**, more memory cannot be allocated at run time if required. Unused allocated **memory is wasted** in case of character array.
- In case of strings, memory is **allocated dynamically**. More memory can be allocated at run time on demand. As no memory is preallocated, **no memory is wasted**.
- Implementation of **character array is faster** than std:: string. **Strings are slower** when compared to implementation than character array.
- Character array **do not offer** much **inbuilt functions** to manipulate strings. String class defines **a number of functionalities** which allow manifold operations on strings.

Practice questions v5

character array and strings

Which of the following declarations are equivalent?

1 and 2 are equivalent

4 and 5 are equivalent

1. **char string_var[10] = "Hello";:** Initializes the string "the easy way."

H	e	l	l	o	\0				
0	1	2	3	4	5	6	7	8	9

Null character automatically appended at the end since it's a string

2. **char string_var[10] = {'H', 'e', 'l', 'l', 'o', '\0'};:** It is equivalent to the first string,

H	e	l	l	o	\0				
0	1	2	3	4	5	6	7	8	9

3. **char string_var[10] = {'H', 'e', 'l', 'l', 'o'};:** It is an array of characters, not a string. It is unique.

H	e	l	l	o					
0	1	2	3	4	5	6	7	8	9

4. **char string_var[6] = "Hello";** It creates a string with room for exactly the 5 characters in "Hello" and the null terminator

H	e	l	l	o	\0
0	1	2	3	4	5

5. **char string_var[] = "Hello";:** It same as creates a string with room for exactly the 5 characters in "Hello" and the null terminator

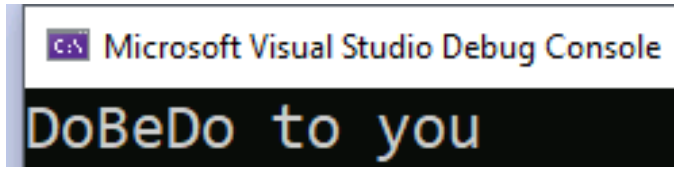
H	e	l	l	o	\0
0	1	2	3	4	5

Explore strcat function and answer the following:

- What string will be stored in singing_string after the following code is run?
  ***char* singing_string[20] = "DoBeDo";**
  strcat(singing_string, " to you");

strcat has some buffer overflow issues..therefore it has been replaced with comparatively secure function " strcat_s"

```
char singing_string[20] = "DoBeDo";
strcat_s(singing_string, " to you"); cout << singing_string;
```



- b

	0	1	2	3	4	5	6	7	8	9
song	I		d	i	d		i	t		\0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
franks_song	I		d	i	d		i	t												

```
#include <iostream>
using namespace std;
int main()
{
    char song[10] = "I did it ";
    char franks_song[20];
    strcpy_s(franks_song, song);
    strcat_s(franks_song, "my way !");
    cout << franks_song << endl;

    return 0;
}
```

Microsoft Visual Studio Debug Console

```
I did it my way!
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
franks_song	I		d	i	d		i	t		m	y		w	a	y	!	\0			

c. **What is the problem (if any) with this code?**

- *char* a_string[20] = "How are you? ";
- strcat(a_string, "Good, I hope.");

H	o	w		a	r	e		y	o	u	?		\0						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

H	o	w		a	r	e		y	o	u	?		G	o	o	d	,	I		Overflow !
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	

What is the maximum length of a string that can be placed in the string variable declared by the following declaration? Explain.

char s[6];

- 5
- Because one cell must be dedicated to null character

How many characters are in each of the following character and string constants?

- a. '\n'

Ans: 1

Since \n is enclosed in single quotes therefore it will be treated as a character (escape sequence)

- b. 'n'

Ans: 1

c. "Mary"

Ans: 5

Since the text is enclosed in double quotes,,thus it will be treated as a sring with null character… 4 letters in the text + 1 for null

d. "M"

Ans: 2

e. "Mary\n"

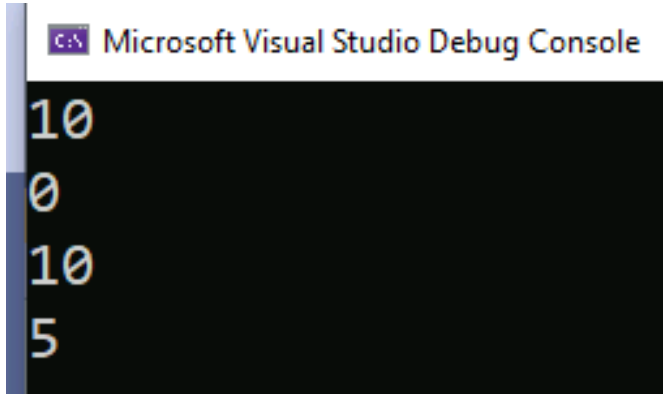
Ans: 6

You can use this script to practice and see the size yourself

Note: the length reported by compiler is “5”. Because the compiler gives the length excluding the null terminator character

M	a	r	y	\n	\0
0	1	2	3	4	5

```
#include <iostream>
using namespace std;
int main()
{
   string s = "Mary\n";
   cout << (int)'\n' << endl;
   cout << (int)s[5] << endl;
   cout << (int)s[4] << endl;
   cout << s.length() << endl;
   return 0;
}
```



Note: 0 is the ASCII code of ‘\0’

5. Given the following declaration and initialization of the string variable, write a loop to assign 'X' to all positions of this string variable, keeping the length the same.

char our_string[15] = "Hi there!";

```
#include <iostream>
using namespace std;
int main()
{
    char ourString[15] = "Hi there!";
    int index = 0;
    while (ourString[index] != '\0')
    {
    ourString[index] = 'X';
    index++;
    }
    return 0;
}
```

6. Consider the following code (and assume it is embedded in a complete and correct program and then run):

```
char my_string[80];
cout << "Enter a line of input:\n";
cin.getline(my_string, 6);
cout << my_string << "<END OF OUTPUT";
```

If the dialogue begins as follows, what will be the next line of output?

Enter a line of input:

May the hair on your toes grow long and curly.

Microsoft Visual Studio Debug Console

```
Enter a line of input:
May the hair on your toes grow long and curly.
May t<END OF OUTPUT
```

M	a	y		t	\0														
0	1	2	3	4	5	6	7	8	9	10	11	12		75	76	77	78	79	

Since inside getline you have mentioned 6, therefore only first 5 characters (+1 for null) will be considered..rest of the input will be ignored

7. Consider the following code:

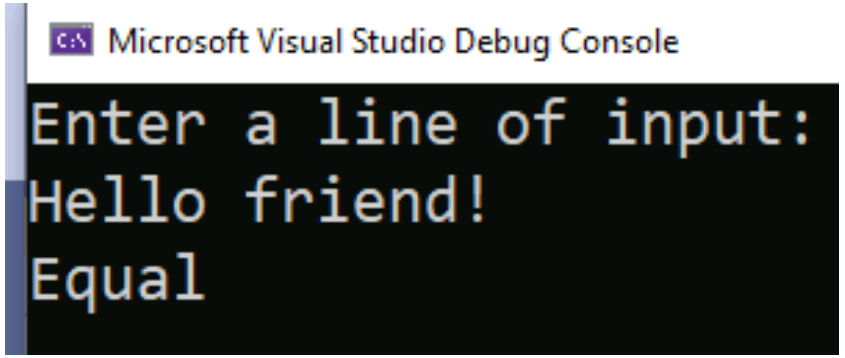
```
string s1, s2("Hello");
cout << "Enter a line of input:\n";
cin >> s1;
if (s1 == s2)
cout << "Equal\n";
else
cout << "Not equal\n";
```

If the dialogue begins as follows, what will be the next line of output?

Enter a line of input:
Hello friend!

1. s2("Hello") is one of the method to initialize a string
2. Since cin takes **space** and **newline** character as a terminating character..therefore, when you entered "Hello Friend!" it will ignore the text after **space.**
3. Therefore, s1="Hello ~~friend!~~"

   i.e. "Hello" which is equal to s2



8. What is the output produced by the following code?

```
string s1, s2("Hello");

s1 = s2;

s2[0] = 'J';

cout << s1 << " " << s2;
```

First character of string gets replaced with ‘J’

Microsoft Visual Studio Debug Console

```
Hello Jello
```

9. Create a Cstring variable that contains a name, age, and title. Each field is separated by a space. For example, the string might contain "Bob 45 Programmer" or any other name/age/title in the same format. Assume the name, age, and title have no spaces themselves. Write a program using only functions from cstring (not the class string) that can extract the name, age, and title into separate variables. Test your program with a variety of names, ages, and titles.

- **Strtok** from cstring library can be used. But, it uses pointers concept so ignore it for now. Let's try doing it without any built-in function
- **Idea:** extract and read the name/age/title and copy them back to individual char arrays
  - Take input using getline since getline can deal with spaces
  - We now have a string that has three words separated by a space
    - Name age title
  - Now we have to iterate the string till we reach \0 i.e till the end.
  - We have to handle following scenarios:
    - When we are going to read the name (no space encountered yet)
    - When we are going to read the age (only one space (first space) has been encountered yet)
    - When we are going to read the title (second space has been encountered)

Microsoft Visual Studio Debug Console

```
Enter the string with name,age,title
Bob 45 Programmer
Extracted name: Bob
Extracted age: 45
Extracted title: Programmer
```

```
#include <iostream>
using namespace std;
int main()
{
    char input[50] = {};
    char name[20] = {};
    char age[5] = {};
    char title[20] = {};

    cout << "Enter the string with name,age,title " <<
    endl;
    cin.getline(input, 50);
    int n_index = 0, a_ind//indexes for three char arrays,
    since we have to start from 0 while setting their
    values
    ex = 0, t_index = 0;
    //indicators to keep track of spaces
    int first_space = 0, second_space = 0;
    for (int i = 0; input[i] != '\0'; i++)
    {
        if ((input[i] != ' '))
        {
            if((first_space == 0) && (second_space == 0))
                name[n_index++] = input[i];
            else if ((first_space == 1) && (second_space
            == 0))
                age[a_index++] = input[i];
            else
                title[t_index++] = input[i];
        }
        if ( (input[i] == ' ') && (first_space == 0) && (
        second_space == 0) )
        {
            first_space = 1;//first space
        encountered, thus now we need to store the age
        }
        else if ((input[i] == ' ') && (first_space == 1)
        && (second_space == 0))
        {
            second_space = 1;//second space
        encountered, thus now we need to store the title
        }
    }
    cout << "Extracted name: "<<name << endl;
    cout << "Extracted age: " << age << endl;
    cout << "Extracted title: " << title << endl;
    return 0;
}
```

10. Write a program that inputs a first and last name, separated by a space, into a string variable. Use the string functions to output the first and last initial. Embed your code into a *do-while* loop. At the end of the loop ask the user if he or she would like to repeat the program. Input the user's choice into a char using cin. If the character is 'y' then repeat the program, otherwise exit. Beware of the pitfall with newlines when cin is mixed with getline

- Two built-in functions can be used:
  - Find:
    - It returns the index of the first occurrence of a character/sub-string
    - **pos = str.find(' ');**

      pos: index of position where ' ' is found in string **str**
    - For further reference, you could see: https://www.geeksforgeeks.org/string-find-in-cpp/
  - Substr:
    - This function takes two values **pos** and **len** as an argument and returns a newly constructed string object with its value initialized to a copy of a sub-string of this object. Copying of string start from *pos* and done till *pos+len* means **[pos, pos+len)**. . e.g
    - **fname= str.substr(0,pos);**
    - For further reference, you could see: https://www.geeksforgeeks.org/substring-in-cpp/

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string str; string fname, lname;
    char c = 'y'; int pos = 0;
    while (c=='y')//to keep on going till the user presses character other than y
    {
        cout << "Enter the text: " << endl;
        getline(cin,str);
        pos = str.find(' '); //search for the position of space character
        cout <<"position of space: " << pos<<endl;
        fname= str.substr(0,pos);
        lname = str.substr(pos+1, str.length());
        cout << "Extracted fname: " << fname<<endl;
        cout << "Extracted lname: " << lname<<endl;
        cout << "Press y if you want to continue " << endl;
        cin >> c;
        cin.ignore(); //As we have already discussed that getline creates mess with
        buffer contents, so it is ideal to clear the buffer before using getline
    }
    return 0;
}
```

11. Write a program that will read in a line of text and output the number of words in the line and the number of occurrences of each letter. Define a word to be any string of letters that is delimited at each end by either whitespace, a period, a comma, or the beginning or end of the line. You can assume that the input consists entirely of letters, whitespace, commas, and periods. When outputting the number of letters that occur in a line, be sure to count upper- and lowercase versions of a letter as the same letter. Output the letters in alphabetical order and list only those letters that do occur in the input line. For example, the input line

I say Hi.

should produce output similar to the following:

3 words
1 a
1 h
2 i
1 s
1 y

4 indicates that alphabet 'a' occurred four times in the text.

- Idea:
  - There are two tasks that must be done:
    - 1) find the number of words (we already covered that in last lecture)
    - 2) find the frequency of each alphabet
  - For (2) idea is to maintain a 26 cells array (for storing the count of each of the 26 alphabets)

4	**1**																								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

  - Assume there are only lowercase alphabets in the input
  - We want to design a logic in which each input character (alphabet) is translated to the values in the range 0,1,…25 (our array index). We will then simply increment that index location.
  - If we closely analyze the ASCII of lowercase alphabets…it starts from 97 (a has 97 ASCII). And we know that we want a to be translated to index 0. Thus we can apply formula:

    index=ASCII value of input character - 97

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
    int letterCount[26] = {};
    string str;
    int i, wrd;
    cout << "Enter the text: " << endl;
    getline(cin,str);
    i = 0;
    wrd = 1;
    while (str[i] != '\0')
    {
        if (str[i] == ' ' || str[i] == '.' ||
        str[i] == ',')
        wrd++;
        i++;
    }
    cout << "Total words: "<<wrd<<endl;
    int alph_index;
    for (int j = 0; j < str.length(); j++)
    {
        if (str[j] != ' ')
        {
            alph_index = str[j] - 97;
            letterCount[alph_index]++;
        }
    }
    cout << "Frequency of alphabets used in
    text: " << endl;
    for (int j = 0; j < 26; j++)
    {
        if (letterCount[j] != 0)
          cout<<"frequency of "<<(char)(j+97)
        <<" is "<<letterCount[j] << endl;
    }
    return 0;
}
```

12. Write a program that reads in a sentence of up to 100 characters and outputs the sentence with spacing corrected and with letters corrected for capitalization. In other words, in the output sentence, all strings of two or more blanks should be compressed to a single blank. The sentence should start with an uppercase letter but should contain no other uppercase letters. Do not worry about proper names; if their first letters are changed to lowercase, that is acceptable. Treat a line break as if it were a blank, in the sense that a line break and any number of blanks are compressed to a single blank. Assume that the sentence ends with a period and contains no other periods. For example, the input

the Answer to life, the Universe, and everything
IS 42.

should produce the following output:

The answer to life, the universe, and everything is 42.

- Idea:
  - Iterate the input character by character and use if statement to do the changes
  - You can write a logic by using ASCII values to go from lowercase to uppercase

13. Write a program that reads in a line of text and replaces all four-letter words with the word "love". For example, the input string

I hate you, you dodo!

should produce the output

I love you, you love!

- Idea:
  - You already practiced how to find the word length and how to replace the characters of your input string
  - Use this knowledge and try to write a program

14. Write a program that reads in a line of text and outputs the line with all the digits in all integer numbers replaced with 'x'. For example,

Input:

My userID is john17 and my 4 digit pin is 1234 which is secret.

Output:

My userID is john17 and my x digit pin is xxxx which is secret

- Idea:
  - You can use the ASCII codes of numeric characters (1,2,3…) to deal with this problem
  - While iterating the input, at any point you can also check the immediate left and right value by using:
    - Arr[i] current character
    - Arr[i+1] right character
    - Arr[i-1] left character

Programming fundamentals

Lecture 13b: Introduction to Functions

Recap

- Char array
- Strings
- Practice questions related to char array and strings

Agenda

- What are functions?
- Built-in vs user defined functions
- Function declaration/definition
- Function signature/Function prototype
- Function invocation/ call
- Returning from a function
- Some examples

What are functions?

- Real world problems are bigger
- Therefore, it is ideal to divide a bigger problem into smaller chunks or components (functions)
- Functions allow you to modularize a program by separating its tasks into self-contained units.
- **Why use functions?**
- Software reuse (no need to repeat the code)
- Divide bigger problem to smaller chunks and easily handle
- Debugging is easy as the program is organized in the form of functions.
  - E.g. if we are looking function findSum then we know that this function will only deal with the logic of finding the sum.

Built-in vs user defined functions

- Built-in function libraries (e.g. the functions that you used from **cmath** and **string** library)
- Functions you write are referred to as user-defined functions. We will learn how to write our own functions soon.

Function	Description	Example
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`

Function prototype

```
int add(int,int);
```

Function definition

```
int add(int a,int b)
{
  return a + b;
}
```

Function call

```
sum = add(1, 2);
```

Function declaration/definition

Function's name

Parameters (comma separated)

Return type

```
int add (int a,int b)
{
    return a + b;
}
```

Function body:
It can contain one or more than one statements

Format of Parameters'list:

Data type
Variable name

Return keyword
This is only needed if we are returning something from a function

Variable/literal/ expression that we are returning from the fuction

What if nothing is returned?
Use **void** keyword

Function's name

Parameters (comma separated)

Return type

```
void add (int a,int b)
{
    cout<<a + b;
}
```

Function body:
It can contain one or more than one statements

Format of Parameters'list:

Data type
Variable name

Nothing is returned in this case, so we can skip RETURN keyword and return parameters

What if nothing is passed as a parameter?
Leave the parameters list empty

Function's name

Return type

```
void add (    )
{
    int a=4,b=5;
    cout<<a + b;
}
```

Function body:
It can contain one or more than one statements

PARAMETERS list will be empty in this case

Example

```
#include <iostream>
using namespace std;
int add(int a,int b)
{
   return a + b;
}
int main()
{
   int a = 5, b = 5;
   int sum = 0;
   sum = add(1, 2);
   sum = add(1, 2);
   cout << "Sum : " << sum <<
   endl;
   return 0;
}
```

We can reuse the function as many times as we want

Function prototype vs function signature

- A **function prototype** (also called a function declaration) tells the compiler the name of a function, the type of data it returns, the number of parameters it expects to receive, the types of those parameters and the order in which the parameters of those types are expected.

```
void maximum( int, int, int );
```

- ***Function Signatures***
- The portion of a function prototype that includes the *name of the function* and the *types of its arguments* is called the function signature or simply the signature. The function signature does not specify the function's return type.

Function prototype

```
int add(int,int);
```

Function definition

```
int add(int a,int b)
{
  return a + b;
}
```

Function call

```
sum = add(1, 2);
```

Inline functions

- Implementing a program as a set of functions is good from a software engineering standpoint, but function calls involve execution-time overhead.
- When a normal function call instruction is encountered, the program stores the memory address of the instructions immediately following the function call statement, loads the function being called into the memory, copies argument values, jumps to the memory location of the called function, executes the function codes, stores the return value of the function, and then jumps back to the address of the instruction that was saved just before executing the called function. **Too much run time overhead**.
- C++ provides inline functions to help reduce function call overhead.

- Placing the qualifier inline before a function's return type in the function definition *advises* the compiler to generate a copy of the function's body code in *every* place where the function is called (when appropriate) to avoid a function call.
- With inline keyword, the compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code. Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program

```
inline return-type function - name(parameters)
{
// function code
}
```

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
   return s * s * s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3) <<
   "\n";
   return 0;
}
```

- **Pros** :-
- It speeds up your program by avoiding function calling overhead.
- It save overhead of variables push/pop on the stack, when function calling happens.
- It save overhead of return call from a function.

Cons :-

- It increases the executable size due to code expansion.

Readings

- Walter Savitch, Problem Solving with C++ The Object of Programming
- Chapter 4, Pg 182

- A good link:
- https://www.youtube.com/watch?v=V9zuox47zr0

Programming fundamentals

Lecture 14: Functions

Recap

- Practice questions related to char array and strings
- Introduction to functions

Agenda

- Functions examples
- Need for function declaration
- Inline functions

What are functions?

- Real world problems are bigger
- Therefore, it is ideal to divide a bigger problem into smaller chunks or components (functions)
- Functions allow you to modularize a program by separating its tasks into self-contained units.
- **Why use functions?**
- Software reuse (no need to repeat the code)
- Divide bigger problem to smaller chunks and easily handle
- Debugging is easy as the program is organized in the form of functions.
  - E.g. if we are looking function findSum then we know that this function will only deal with the logic of finding the sum.

Function declaration/definition

Function's name

Parameters (comma separated)

Return type

```
int add (int a,int b)
{
    return a + b;
}
```

Function body:
It can contain one or more than one statements

Format of Parameters'list:

Data type
Variable name

Return keyword
This is only needed if we are returning something from a function

Variable/literal/ expression that we are returning from the fuction

What if nothing is returned?
Use **void** keyword

Function's name

Parameters (comma separated)

Return type

```
void add (int a,int b)
{
    cout<<a + b;
}
```

Function body:
It can contain one or more than one statements

Format of Parameters'list:

Data type
Variable name

Nothing is returned in this case, so we can skip RETURN keyword and return parameters

What if nothing is passed as a parameter?
Leave the parameters list empty

Function's name

Return type

```
void add (    )
{
    int a=4,b=5;
    cout<<a + b;
}
```

Function body:
It can contain one or more than one statements

PARAMETERS list will be empty in this case

Example

```
#include <iostream>
using namespace std;
int add(int a,int b)
{
   return a + b;
}
int main()
{
   int sum = 0;
   sum = add(1, 2);
   sum = add(1, 2);
   cout << "Sum : " << sum <<
   endl;
   return 0;
}
```

We can reuse the function as many times as we want

Function prototype vs function signature

- A **function prototype** (also called a function declaration) tells the compiler the name of a function, the type of data it returns, the number of parameters it expects to receive, the types of those parameters and the order in which the parameters of those types are expected.

```
void maximum( int, int, int );
```

- ***Function Signatures***
- The portion of a function prototype that includes the *name of the function* and the *types of its arguments* is called the function signature or simply the signature. The function signature does not specify the function's return type.

Function prototype

```
int add(int,int);
```

Function definition

```
int add(int a,int b)
{
  return a + b;
}
```

Function call

```
sum = add(1, 2);
```

Example

```
   #include <iostream>
   using namespace std;
1.1 int add(int x, int y)
   //definition
1.2 {
       return x + y;
   }
   int main()
   {1
    2 int a = 5, b = 6;
    3 int sum = 0;
    4 sum=add(a,b);//call
    5 cout << "Sum : " << sum <<
      endl;
      return 0;
   }
```

Line#	a	b	sum	Add func call	x	y
1	5	6	-	-	-	-
2	5	6	0	-	-	-
3	5	6	0	add(5,6)	-	-
1.1	5	6	0		5	6
1.2	5	6	0	Go to step 3	5	6
3	5	6	11		-	-
4	5	6	11	***Print sum to the console***		

sum	**11**	**x**	**5**	**y**	**6**	**a**	**5**	**b**	**6**
	0XF621		**0XF001**		**0XFE10**		**0FF002**		**0FF512**

Function call works like a go to statement (from program execution perspective). It switches the flow of program to a specific memory cell. Otherwise, as you know, the program gets executed line wise. But function call preempts the normal flow and directs the control to the specific line#.

This is how your code is organized in memory, for execution.

0XEA00	`int main(){`
0XEA01	int a = 5, b = 6;
0XEA02	int sum = 0;
0XEA03	sum=add(a,b);
0XEA04	cout << "Sum: " << sum << endl;
0XEA05	return 0;}
0XEA06	

Function call transfers the control to the specific line where we have function definition (cell address 0YYA54)

0YYA54	int add(int x, int y){
0YYA55	return x + y;}
0YYA56	

Memory addresses

sum	11	x	5	y	6	a	5	b	6
	0XF621		0XF001		0XFE10		0FF002		0FF512

What if the name of variable passed as an argument in function call and variable mentioned in function argument is same?

- Ignore it for now. We will see what happens in this scenario in next lecture when we will cover scope, local and global variables. For now, use different names to avoid confusion during practice.

```
#include <iostream>
using namespace std;
int add(int a, int b)//definition
{
        return a + b;
}
int main()
{
    int a = 5, b = 6;
    int sum = 0;
    sum=add(a,b);//call
    sum = add(1, 2);//call
    cout << "Sum : " << sum << endl;
    return 0;
}
```

variable mentioned in function argument

Same name is given to both i.e. a and b

variable passed as an argument in function call

Inline functions

- Implementing a program as a set of functions is good from a software engineering standpoint, but function calls involve execution-time overhead.
- When a normal function call instruction is encountered, the program stores the memory address of the instructions immediately following the function call statement, loads the function being called into the memory, copies argument values, jumps to the memory location of the called function, executes the function codes, stores the return value of the function, and then jumps back to the address of the instruction that was saved just before executing the called function. **Too much run time overhead**.
- C++ provides inline functions to help reduce function call overhead.

- Placing the qualifier inline before a function's return type in the function definition *advises* the compiler to generate a copy of the function's body code in *every* place where the function is called (when appropriate) to avoid a function call.
- With inline keyword, the compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code. Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program

```
inline return-type function - name(parameters)
{
// function code
}
```

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
   return s * s * s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3) <<
   "\n";
   return 0;
}
```

- **Pros** :-
- It speeds up your program by avoiding function calling overhead.
- It save overhead of variables push/pop on the stack, when function calling happens.
- It save overhead of return call from a function.

Cons :-

- It increases the executable size due to code expansion.

Readings

- Walter Savitch, Problem Solving with C++ The Object of Programming
- Chapter 4, Pg 182

- A good link:
- https://www.youtube.com/watch?v=V9zuox47zr0

Programming fundamentals

Lecture 15: Scope, *Local/global variables* and *Pass by value in functions*

Recap

- Functions examples
- Need for function declaration
- Function prototype and signature
- Function declaration/definition/call

Agenda

- Inline functions
- Some examples of functions
- Scope
- Local and global variables
- Pass by value

```
   #include <iostream>
   using namespace std;
1.1 int add(int x, int y)
   //definition
1.2 {
       return x + y;
   }
   int main()
   {1
    2 int a = 5, b = 6;
    3 int sum = 0;
    4 sum=add(a,b);//call
    5 cout << "Sum : " << sum <<
      endl;
      return 0;
   }
```

Line#	a	b	sum	Add func call	x	y
1	5	6	-	-	-	-
2	5	6	0	-	-	-
3	5	6	0	add(5,6)	-	-
1.1	5	6	0		5	6
1.2	5	6	0	Go to step 3	5	6
3	5	6	11		-	-
4	5	6	11	***Print sum to the console***		

sum	**11**	**x**	**5**	**y**	**6**	**a**	**5**	**b**	**6**
	0XF621		**0XF001**		**0XFE10**		**0FF002**		**0FF512**

Inline functions

- Implementing a program as a set of functions is good from a software engineering standpoint, but function calls involve execution-time overhead.
- When a normal function call instruction is encountered, the program stores the memory address of the instructions immediately following the function call statement, loads the function being called into the memory, copies argument values, jumps to the memory location of the called function, executes the function codes, stores the return value of the function, and then jumps back to the address of the instruction that was saved just before executing the called function. **Too much run time overhead**.
- C++ provides inline functions to help reduce function call overhead.

- Placing the qualifier inline before a function's return type in the function definition *advises* the compiler to generate a copy of the function's body code in *every* place where the function is called (when appropriate) to avoid a function call.
- With inline keyword, the compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code. Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program

```
inline return-type function - name(parameters)
{
// function code
}
```

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
   return s * s * s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3) <<
   "\n";
   return 0;
}
```

- **Pros** :-
- It speeds up your program by avoiding function calling overhead.
- It save overhead of variables push/pop on the stack, when function calling happens.
- It save overhead of return call from a function.

Cons :-

- It increases the executable size due to code expansion.

Readings

- Walter Savitch, Problem Solving with C++ The Object of Programming
- Chapter 4, Pg 182

- A good link:
- https://www.youtube.com/watch?v=V9zuox47zr0

Example#1: Find sum and average of two numbers using separate functions

```
#include<iostream>
using namespace std;
int FindSum(int x,int y)
{
return x + y;
}

double FindAvg(double s)
{
return s / 2;
}
```

```
int main()
{
    //declare variables
    int sum = 0;double avg = 0;
    int a=0, b=0;
    //take input
    cin >> a >> b;
    //call FindSum function and pass a
    and b as arguments
    sum = FindSum(a,b);
    //call FindAvg function and pass
    sum as an argument
    avg = FindAvg(sum);
    cout << "Sum : " << sum<<endl;
    cout << "Average : " << avg << endl;
    return 0;
}
```

Contd..

```
//we can also do it as:
// avg=FindSum(a,b)/2;
//in that case there is no
need for FindAvg function
```

```
#include<iostream>
using namespace std;
int FindSum(int x,int y)
{
  return x + y;
}

int FindAvg(double s)
{
  return s / 2;
}
```

```
int main()
{
    //declare variables
    int sum = 0;double avg = 0;
    int a=0, b=0;
    //take input
    cin >> a >> b;
    //call FindSum function and pass a
    and b as arguments
    sum = FindSum(a,b);
    //call FindAvg function and pass
    sum as an argument
    avg = FindAvg(sum);
    cout << "Sum: " << sum<<endl;
    cout << "Average: " << avg << endl
    return 0;
}
```

If we change **double** to **int**, then no error will occur. Compiler will do implicit typecasting that we discussed in earlier lectures

Example # 2:

Write a program that implements a function that declares an array of 4 elements and take their sum. After that if returns

1 if the sum lies in the range 0-10

2 if the sum lies in the range 11-30

3 if the sum is greater than 30

The relevant message regarding the range should be printed on console in the main function

```
#include<iostream>
using namespace std;
int findSumOfElements()
{
    int arr[4] = {1,3,2,10};
    int sum = 0;
    for (int i=0;i<4;i++)
    {
          sum = sum + arr[i];
    }
    if (sum >= 0 && sum <= 10)
          return 1;
    else if (sum >= 11 && sum <= 30)
          return 2;
    else
          return 3;
}
```

```
int main()
{
    int flag = 0;
    flag=findSumOfElements();
    if (flag == 1)
      cout << "Sum lies in the range 0-10";
    else if(flag==2)
      cout << "Sum lies in the range 11-30";
    else
      cout << "Sum is greater than 30";
    return 0;
}
```

Example#3: Keep on taking input from the user in an array till the user enters any character other than ‘y’. Implement a separate function to take input in an array.

```
#include<iostream>
using namespace std;
void takeInput()
{
   int arr[4];
   cout << "Enter the values for array"
   << endl;
   for (int i=0;i<4;i++)
   cin >> arr[i];
   for (int i = 0; i < 4; i++)
   cout << arr[i]<<" ";
}
```

```
int main()
{
   char ch = 'y';
   while (ch == 'y')
   {
      takeInput();
      cout << endl;
      cout << "Enter y if you want to
      take input again?" << endl;
      cin >> ch;
   }
   return 0;
}
```

Procedural abstraction/function abstraction/information hiding

- Abstracting away the details of the code contained in the function body.
- Also known as *the black-box principle* .. means that your function should be written so that it can be used like a **black box.**
- This means that the programmer who uses the function should not need to look at the body of the function definition

Scope

- Scope of a variable is defined as the extent of the program code within which the variable can we accessed or declared or worked with. There are mainly two types of variable scopes:
  - Local Variables
    - Variables defined within a function or block are said to be **local** to those functions. Anything between '{' and '}' is said to inside a block.
  - Global Variables
    - Global Variables can be accessed from any part of the program.
    - They are available through out the life time of a program.
    - They are declared at the top of the program outside all of the functions or blocks.

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

Global Variable

```
// global variable
int global = 5;

// main function
int main()
{
    // local variable with same
    // name as that of global variable
    int global = 2;

    cout << global << endl;
}
```

Local variable

Some basic rules

Local variables

- Variables within one scope cannot have similar names.
- E.g. the following code will give an error

```
#include<iostream>
using namespace std;
void takeSum(int a,int b)
{
 int a;
}
int main()
{
   takeSum(1,2);
   return 0;
}
```

scope

Error List

Entire Solution | 1 Error | 0 Warnings

	Code	Description
	C2082	redefinition of formal parameter 'a'

What if same names are used in separate functions

- Variable name in separate functions might be same.
- E.g. a is both declared and used in **takeSum** and **main** function

a	2	0XF621
b	4	0XF231

a	5	0XF880
b	3	0XF222

```
#include<iostream>
using namespace std;
void takeSum(int a,int b)
{
    cout << "Sum : "<<a + b;
}

int main()
{
    takeSum(2,4);
    int a = 5,b=3;
    cout << "Main : " << a << " " << b;
    return 0;
}
```

Scope of variables declared in takeSum

Scope of variables a and b

Microsoft Visual Studio Debug Console

```
Sum: 6
Main: 5 3
```

Some basic rules

Global variables

Global variable

- Global variables are defined outside of all the functions, usually on top of the program.
- The global variables will hold their value throughout the lifetime of your program.
- A global variable can be accessed by any function.
- That is, a global variable is available for use throughout your entire program after its declaration.

Example

```
   #include<iostream>
   using namespace std;
1  void print(double);
2  double tax_rate=0;
3  int main()
4  {
5  cout << "Welome to banking application" << endl;
6  cout << "Enter the tax rate in percentage: " << endl;
7  cin >> tax_rate;
8  double balance = 2000;
9  double deduction = balance * (tax_rate)/100;
10 print(balance - deduction);
11 return 0;
12 }
13 void print(double b)
14 {
15 cout << "Final balance: " << b << endl;
16 cout << "tax rate applied: " << tax_rate;
17 }
```

- Global variable named **tax_rate** is declared, outside all the functions.
- Its scope will start from the point of declaration (Line#2) and end till the end of your program (i.e. last line of program.. line#17)

Shadowing

- local variables with the same name as a global variable will shadow the global variable wherever the local variable is in scope
- Value of global variable will then be **hidden** and local value persists till the scope of the local variable

```
#include<iostream>
using namespace std;
void print(double);
double tax_rate=0;
int main()
{
    int tax_rate;
    cout << "Welome to banking application" << endl;
    cout << "Enter the tax rate in percentage: " << endl;
    cin >> tax_rate;
    double balance = 2000;
    double deduction = balance * (tax_rate)/100;
    print(balance - deduction);
    return 0;
}
void print(double b)
{
    cout << "Final balance: " << b << endl;
    cout << "tax rate applied: " << tax_rate;
}
```

Tax_rate

0
0XF621

Tax_rate

Garbage
0XF666

Microsoft Visual Studio Debug Console

```
Welome to banking application
Enter the tax rate in percentage:
10
Final balance: 1800
tax rate applied: 0
```

Scope of variables

```
   #include<iostream>
   using namespace std;
1  void print(double);
2  double tax_rate=0;
3  int main()
4  {
5      int tax_rate;
6      cout << "Welome to banking application" << endl;
7      cout << "Enter the tax rate in percentage: " << endl;
8      cin >> tax_rate;
9      double balance = 2000;
10     double deduction = balance * (tax_rate)/100;
11     print(balance - deduction);
12     return 0;
13 }
14 void print(double b)
15 {
16     cout << "Final balance: " << b << endl;
17     cout << "tax rate applied: " << tax_rate;
18 }
```

Tax_rate

0
0XF621

Tax_rate

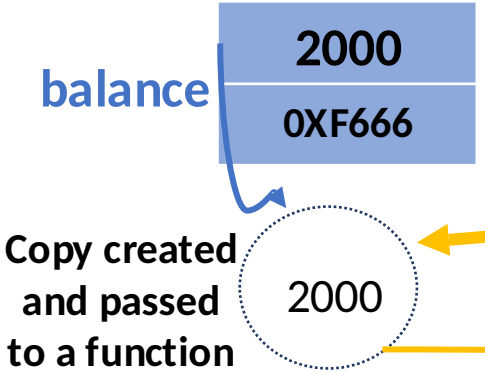
Garbage
0XF666

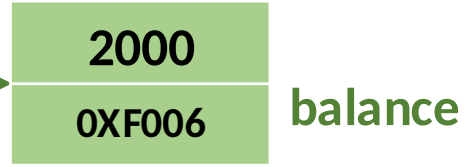
Variables	Scope	
	From Line#	To Line#
tax_rate	2	18
tax_rate	5	13
balance	9	13
deduction	10	13
b	14	18

Pass by value

- While calling the function with certain arguments, only the copy of arguments gets passed.
- The original value of the argument remains same. Its copy is created and passed.

```
#include<iostream>
using namespace std;
void print(double);
int main()
{
    double balance = 2000;
    print(balance);
    cout << "Balance (in main function): " << balance << en
    return 0;
}
void print(double balance)
{
    balance -= 1000;
    cout << "Balance (in print function): " << balance << e
    ;
}
```





Change will occur in local variable of ***print*** function. i.e. at location **0XF006**

Microsoft Visual Studio Debug Console

```
Balance (in print function): 1000
Balance (in main function): 2000
```

Programming fundamentals

Lecture 16: *Pass by value* vs *Pass by reference* in functions

Recap

- Inline functions
- Some examples of functions
- Scope
- Local and global variables

Agenda

- Shadowing
- Pass by value
- Pass by reference
- Arrays with functions

Shadowing

- local variables with the same name as a global variable will shadow the global variable wherever the local variable is in scope
- Value of global variable will then be **hidden** and local value persists till the scope of the local variable

```
#include<iostream>
using namespace std;
void print(double);
double tax_rate=0;
int main()
{
    int tax_rate;
    cout << "Welome to banking application" << endl;
    cout << "Enter the tax rate in percentage: " << endl;
    cin >> tax_rate;
    double balance = 2000;
    double deduction = balance * (tax_rate)/100;
    print(balance - deduction);
    return 0;
}
void print(double b)
{
    cout << "Final balance: " << b << endl;
    cout << "tax rate applied: " << tax_rate;
}
```

Tax_rate

0
0XF621

Tax_rate

Garbage
0XF666

Microsoft Visual Studio Debug Console

```
Welome to banking application
Enter the tax rate in percentage:
10
Final balance: 1800
tax rate applied: 0
```

Scope of variables

```
#include<iostream>
using namespace std;
1 void print(double);
2 double tax_rate=0;
3 int main()
4 {
5     double tax_rate;
6     cout << "Welome to banking application" << endl;
7     cout << "Enter the tax rate in percentage: " << endl;
8     cin >> tax_rate;
9     double balance = 2000;
10    double deduction = balance * (tax_rate)/100;
11    print(balance - deduction);
12    return 0;
13 }
14 void print(double b)
15 {
16    cout << "Final balance: " << b << endl;
17    cout << "tax rate applied: " << tax_rate;
18 }
```

Tax_rate

0
0XF621

Tax_rate

Garbage
0XF666

Variables	Scope	
	From Line#	To Line#
tax_rate	2	18
tax_rate	5	13
balance	9	13
deduction	10	13
b	14	18

Basic rule

A. Two local variables within a same scope **cannot** have a same name

B. Two variables (one local and one global) **can** have a same name. (shadowing concept will be applied)

Rule A

```
#include<iostream>
using namespace std;
void print(double b)
{
     double b;
}
int main()
{
   int tax_rate;
   cin >> tax_rate;
   int tax_rate;
   return 0;
}
```

Local variables with same name as "b" cannot be declared in one scope

Local variables with same name as "tax_rate" cannot be declared in one scope

Rule B

There can be two variables **can** have a same name " tax_rate", provided that one is local and the other is global. *(shadowing concept will be applied)*

```
#include<iostream>
using namespace std;
void print(double);
double tax_rate=0;
int main()
{
    double tax_rate;
    cout << "Welome to banking application" << endl;
    cout << "Enter the tax rate in percentage: " << endl
    cin >> tax_rate;
    double balance = 2000;
    double deduction = balance * (tax_rate)/100;
    print(balance - deduction);
    return 0;
}
void print(double b)
{
    cout << "Final balance: " << b << endl;
    cout << "tax rate applied: " << tax_rate;
}
```

Pass by value

- While calling the function with certain arguments, only the copy of arguments gets passed.
- The original value of the argument remains same. Its copy is created and passed.
- Pass by value is used when the function does not want to modify the parameter

```
#include<iostream>
using namespace std;
void print(double);
int main()
{
    double balance = 2000;
    print(balance);
    cout << "Balance (in main function): " << balance << en
    return 0;
}
void print(double balance)
{
    balance -= 1000;
    cout << "Balance (in print function): " << balance << e
    ;
}
```

2000	
0XF666	

balance

Copy created and passed to a function

2000

2000	
0XF006	

balance

Change will occur in local variable of ***print*** function. i.e. at location **0XF006**

Microsoft Visual Studio Debug Console

```
Balance (in print function): 1000
Balance (in main function): 2000
```

Pass by reference

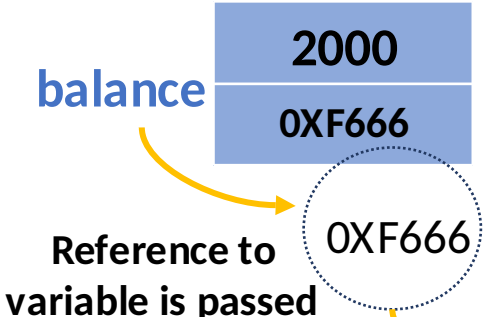
- Pass by reference is used when the function wants to modify the actual parameter
- Append the **ampersand sign &** to its type name.
- The corresponding argument in a call to the function should then be a variable, not a constant or other expression.
- When the function is called, the corresponding variable argument (not its value) will be substituted for the formal parameter (local variable created in parameters list).
- Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

Syntax

```
#include<iostream>
using namespace std;
void print(double &);
int main()
{
    double balance = 2000;
    print(balance);
    cout << "Balance (in main
    function): " << balance << endl;
    return 0;
}
void print(double &balance)
{
    balance -= 1000;
    cout << "Balance (in print
    function): " << balance << endl;
}
```

Mention **&** in function declaration *and* header!

Not in function call!



Reference to variable is passed No separate variable is created. Original variable is referenced

```
#include<iostream>
using namespace std;
void print(double &);
int main()
{
    double balance = 2000;
    print(balance);
    cout << "Balance (in main
    function): " << balance << endl;
    return 0;
}
void print(double &balance)
{
    balance -= 1000;
    cout << "Balance (in print
    function): " << balance << endl;
}
```

Change will occur in actual(original) variable whose reference is passed. i.e. at location **0XF666**

Microsoft Visual Studio Debug Console

```
Balance (in print function): 1000
Balance (in main function): 1000
```

Swap
(pass by value)

```
#include<iostream>
using namespace std;
void swap(int a,int b)
{
   int temp=0;
   temp = a;
   a = b;
   b = temp;
}
```

```
int main()
{
   int a = 3, b = 5;
   cout << "Values before swapping: " <<
   endl;
   cout << a << " " << b<<endl;
   swap(a, b);
   cout << "Values after swapping: " <<
   endl;
   cout << a << " " << b << endl;
   return 0;
}
```

Microsoft Visual Studio Debug Console

```
Values before swapping:
3 5
Values after swapping:
3 5
```

Swap
(pass by reference)

```
#include<iostream>
using namespace std;
void swap(int &a,int &
b)
{
   int temp=0;
   temp = a;
   a = b;
   b = temp;
}
```

```
int main()
{
   int a = 3, b = 5;
   cout << "Values before swapping: " <<
   endl;
   cout << a << " " << b<<endl;
   swap(a, b);
   cout << "Values after swapping: " <<
   endl;
   cout << a << " " << b << endl;
   return 0;
}
```

Microsoft Visual Studio Debug Console

```
Values before swapping:
3 5
Values after swapping:
5 3
```

Home task

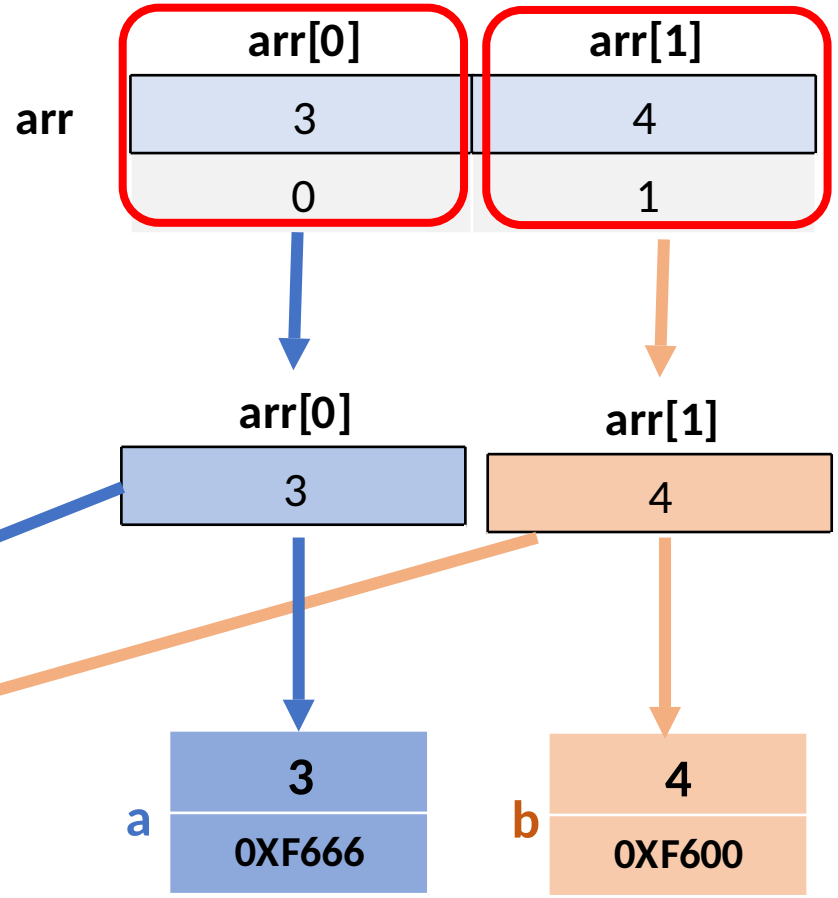
Write a function that swap two values without using any third variable.

Hint: You might need to write a mathematical expression that finds sum of values and then subtract one of the values from this sum to get the other one

Arrays with functions

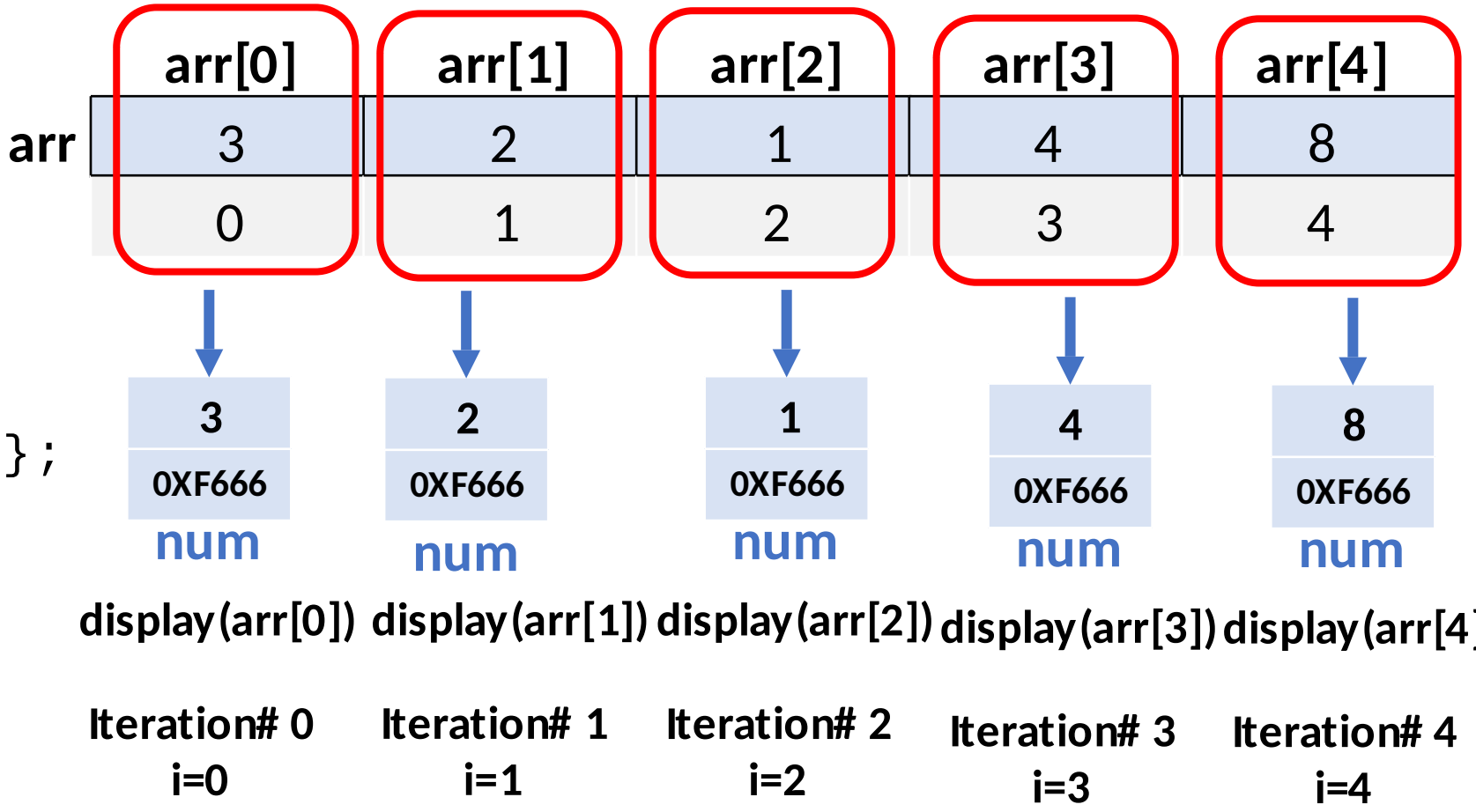
Passing a ***specific cell value of an array*** to a function

```
#include<iostream>
using namespace std;
int add(int a, int b)
{
  return a + b;
}
int main()
{
   int arr[2] = { 3,4 };
   int sum=add(arr[0], arr[1]);
   cout << "Sum :" <<sum<< endl;
   return 0;
}
```



Passing a ***specific cells value of an array*** using loop

```
#include<iostream>
using namespace std;
void display(int num)
{
    cout << num <<endl;
}
int main()
{
   int arr[5] = {3,2,1,4,8};
   for (int i = 0; i < 5;
   i++)
       display(arr[i]);
   return 0;
}
```



Entire Arrays as Function Arguments

- If we want to pass a complete array as an argument then following rules must be followed:
  - In function header, generally we pass two values: array (mandatory) and its size (optional)
  - Mention square brackets after array name

```
void display(int num[], int size)
```

  - In function call, don't mention square brackets,
  - Just mention array name and size.

```
display(arr,5);
```

```
#include<iostream>
using namespace std;
void display(int num[], int size)
{
    for (int i = 0; i < 5; i++)
        cout << num[i] <<endl;
}
int main()
{
    int arr[5] = {3,2,1,4,8};
    display(arr,5);
    return 0;
}
```

```
#include<iostream>
using namespace std;
void display(int num[], int size)
{
    cout << "Address of num:" << endl;
    for (int i = 0; i < size; i++)
       cout << &num[i]<<" ";
    cout << endl;
    for (int i = 0; i < size; i++)
       cout << num[i] <<endl;
}
int main()
{
    int arr[5] = {3,2,1,4,8};
    cout << "Address of arr:" << endl;
    for (int i = 0; i < 5; i++)
       cout << &arr[i] << " ";
    cout << endl;
    display(arr,5);
    return 0;
}
```

	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
arr	3	4	1	4	8
	0	1	2	3	4
	00DX0	***00DX1***	***00DX2***	***00DX3***	***00DX4***

Array **arr** copied to **num** (like by reference) . That is, the actual array gets passed.

You can notice that addresses of both arrays are same. Because no copy will be created. Original array will be transferred as it is

	num[0]	num[1]	num[2]	num[3]	num[4]
num	3	4	1	4	8
	0	1	2	3	4
	00DX0	***00DX1***	***00DX2***	***00DX3***	***00DX4***

- Passing size is optional. E.g. you may skip passing the size:
- Note: you may pass as many arguments along with an array as you want. There is no restriction that only array and its size must be passed.

```
#include<iostream>
using namespace std;
void display(int num[])
{
   cout << "Address of num:" << endl;
   for (int i = 0; i < 5; i++)
     cout << &num[i]<<" ";
   cout << endl;
   for (int i = 0; i < 5; i++)
     cout << num[i] <<endl;
}
```

```
int main()
{
   int arr[5] = {3,2,1,4,8};
   cout << "Address of arr:" << endl;
   for (int i = 0; i < 5; i++)
     cout << &arr[i] << " ";
   cout << endl;
   display(arr);
   return 0;
}
```

Programming fundamentals

Lecture 17: *Pass 2D array to a function, static variables, function overloading, pointers*

Recap

- Shadowing
- Pass by value
- Pass by reference
- Arrays with functions

Scope of variables

```
#include<iostream>
using namespace std;
1 void print(double);
2 double tax_rate=0;
3 int main()
4 {
5     double tax_rate;
6     cout << "Welome to banking application" << endl;
7     cout << "Enter the tax rate in percentage: " << endl;
8     cin >> tax_rate;
9     double balance = 2000;
10    double deduction = balance * (tax_rate)/100;
11    print(balance - deduction);
12    return 0;
13 }
14 void print(double b)
15 {
16    cout << "Final balance: " << b << endl;
17    cout << "tax rate applied: " << tax_rate;
18 }
```

Tax_rate

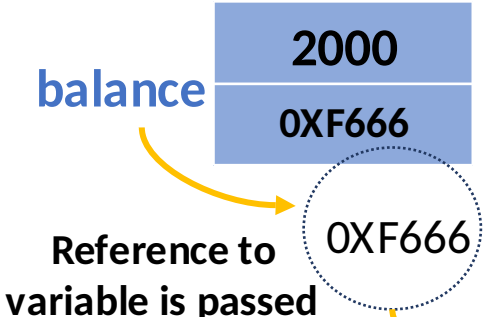
0
0XF621

Tax_rate

Garbage
0XF666

Variables	Scope	
	From Line#	To Line#
tax_rate	2	18
tax_rate	5	13
balance	9	13
deduction	10	13
b	14	18

Pass by reference



Reference to variable is passed No separate variable is created. Original variable is referenced

```
#include<iostream>
using namespace std;
void print(double &);
int main()
{
    double balance = 2000;
    print(balance);
    cout << "Balance (in main
    function): " << balance << endl;
    return 0;
}
void print(double &balance)
{
    balance -= 1000;
    cout << "Balance (in print
    function): " << balance << endl;
}
```

Change will occur in actual(original) variable whose reference is passed. i.e. at location **0XF666**

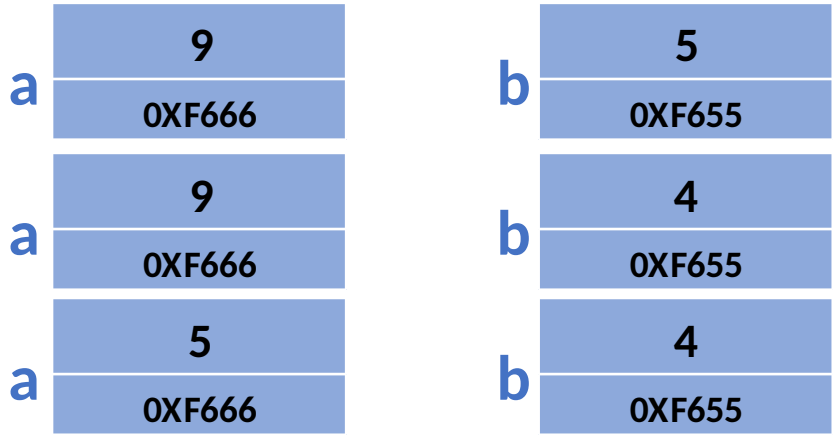
Microsoft Visual Studio Debug Console

```
Balance (in print function): 1000
Balance (in main function): 1000
```

Home task

Write a function that swap two values without using any third variable.

```
#include<iostream>
using namespace std;
int main() {
    int a = 4, b = 5;
    cout <<"Before swapping:"<< endl;
    cout << a << " " << b << " " <<
    endl;

    a = a + b;//9

    b = a - b;//9-5=4

    a = a - b;//9-4=5
    cout << "After swapping:" << endl;
    cout << a << " " << b << " " <<
    endl;
    return 0;
```



Agenda

- Passing 2D arrays
- Static variables
- Function overloading
- Introduction to pointers

Pass 2D array to a function

- Syntax is slightly different from the one we studied for 1D array
- first array dimension does not have to be specified.
- The second (and any subsequent) dimensions must be given

```
#include<iostream>
using namespace std;
void print(int arr[2][3])
{
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
            cout << arr[i][j]<<" ";
        cout << endl;
    }
}
void print1D(int arr[])
{
    for (int j = 0; j < 2; j++)
        cout << arr[j] << " "
    cout << endl;
}
```

```
void print(int arr[][3])
```

You can skip first dimension size and keep the first bracket empty..but subsequent brackets MUST be filled.

```
int main()
{
    int arr1[2] = { 1, 2 };
    print1D(arr1);

    int arr[2][3] = { {1, 2, 3}, {4, 5, 6}
    print(arr);
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
1 2
1 2 3
4 5 6
```

arr1

1	2
0XF666	0XF670

arr

0XF500	0XF504	0XF508
1	2	3
4	5	6
0XF512	0XF516	0XF520

Error ! You cannot leave both brackets empty . while passing 2D array

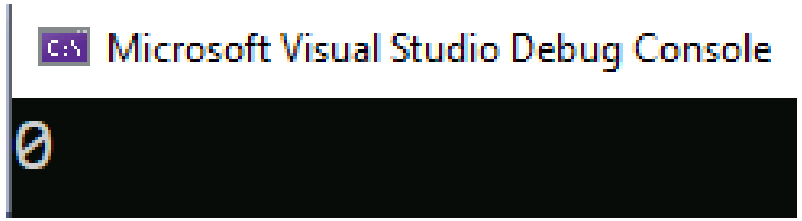
```
void print(int arr[][])
{
   for (int i = 0; i < 2; i++)
   {
      for (int j = 0; j < 3; j++)
            cout << &arr[i][j]<<" ";
      cout << endl;
   }
}
```

Static variables

- When a variable is declared as static, space for **it gets allocated for the lifetime of the program**
- Static variables when used inside function are initialized only once, and then they hold there value even through function calls.

static int count=0;

- An uninitialized static variable set to **0**.




```
#include<iostream>
using namespace std;
int main()
{
    static int s;
    cout << s  << endl;
    return 0;
}
```

Without using static

```
#include<iostream>
using namespace std;
void counter()
{
    int count = 0;
    cout << count++;
}

int main()
{
    for (int i = 0; i < 5; i++)
    {
        counter();
    }
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
00000
```

If we do not use static keyword, the variable count, is reinitialized every time when counter() function is called, and gets destroyed each time when counter() functions ends. But, if we make it static, once initialized count will have a scope till the end of main() function and it will carry its value through function calls too.

With using static

```
#include<iostream>
using namespace std;
void counter()
{
    static int count = 0;
    cout << count++;
}

int main()
{
    for (int i = 0; i < 5; i++)
    {
        counter();
    }
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
01234
```

Example

```
#include <iostream>
using namespace std;
void tryMe(int& v);
int main()
{
    int x = 8;
    for (int count = 1; count < 5;
    count++)
        tryMe(x);
    return 0;
}
```

```
void tryMe(int& v)
{
    static int num = 2;
    if (v % 2 == 0)
    {
        num++;
        v = v + 3;
    }
    else
    {
        num--;
        v = v + 5;
    }
    cout << v << ", " << num << end
}
```

Microsoft Visual Studio Debug Console

```
11, 3
16, 2
19, 3
24, 2
```

Function overloading

- Function Overloading in C++ can be defined as the process of having two or more member functions of a class with the same name, but different in parameters.
- In function overloading, the function can be redefined either 1) **by using different types of arguments** or 2) **a different number of arguments according to the requirement**. Return type is irrelevant
- It is only through these differences compiler can differentiate between the two overloaded functions.
- One of the major advantages of Function overloading is that it increases the readability of the program because we don't need to use different names for the same action again and again.

- **Design Calculator to add and multiply 2 and 3 int, double, long, short numbers**

```
#include <iostream>
using namespace std;
void addInt_2(int a,int b);
void addInt_3(int a, int b, int c);
void addShort_2(short a, short b);
void addShort_3(short a, short b, short c);
void addLong_2(long a, long b);
void addLong_3(long a, long b, long c);
void addDouble_2(double a, double b);
void addDouble_3(double a, double b, double c);
int main()
{
   return 0;
}
void addInt_2(int a, int b)
{
   cout << "Sum : "<<a + b<<endl;
}
void addInt_3(int a, int b, int c)
{
   cout << "Sum : " << a + b + c << endl;
}
void addShort_2(short a, short b)
{
   cout << "Sum : " << a + b << endl;
}
void addShort_3(short a, short b, short c)
{
   cout << "Sum : " << a + b + c << endl;
}
void addLong_2(long a, long b)
{
   cout << "Sum : " << a + b << endl;
}
void addLong_3(long a, long b, long c)
{
   cout << "Sum : " << a + b + c << endl;
}
void addDouble_2(double a, double b)
{
   cout << "Sum : " << a + b << endl;
}
void addDouble_3(double a, double b, double c)
{
   cout << "Sum : " << a + b + c << endl;
}
```

```
void addInt_2(int a,int b);
void addInt_3(int a, int b, int c);
void addShort_2(short a, short b);
void addShort_3(short a, short b, short c
);
void addLong_2(long a, long b);
void addLong_3(long a, long b, long c);
void addDouble_2(double a, double b);
void addDouble_3(double a, double b,
double c);
```

Problems:

- Code is not well readable and understandable (from func calls)
- We have to remember name of various versions of add functions
- Difficult to debug

Solution

Function overloading: using same name (e.g. **add**) for all versions of a function

With function overloading

```
#include <iostream>
using namespace std;
void add(int a, int b);
void add(int a, int b, int c);
void add(short a, short b);
void add(short a, short b, short c);
void add(long a, long b);
void add(long a, long b, long c);
void add(double a, double b);
void add(double a, double b, double c);
int main()
{
return 0;
}
void add(int a, int b)
{
cout << "Sum : "<<a + b<<endl;
}
void add(int a, int b, int c)
{
cout << "Sum : " << a + b + c << endl;
}
void add(short a, short b)
{
cout << "Sum : " << a + b << endl;
}
```

```
void add(short a, short b, short c)
{
cout << "Sum : " << a + b + c << endl;
}
void add(long a, long b)
{
cout << "Sum : " << a + b << endl;
}
void add(long a, long b, long c)
{
cout << "Sum : " << a + b + c << endl;
}
void add(double a, double b)
{
cout << "Sum : " << a + b << endl;
}
void add(double a, double b, double c)
{
cout << "Sum : " << a + b + c << endl;
}
```

```
void add(int a, int b);
void add(int a, int b, int c);
void add(short a, short b);
void add(short a, short b, short c);
void add(long a, long b);
void add(long a, long b, long c);
void add(double a, double b);
void add(double a, double b, double c);
```

The functions must differ with respect to either: 1) type of parameter(s) or 2) count of parameters.

Return type can be same or different.

```
#include <iostream>
using namespace std;
void add(int a,int b);
int add(int a, int b);
int main
{
    retu
}
void add
```

void add(int a, int b)
+7 overloads
Search Online

cannot overload functions distinguished by return type alone

Search Online

If only return type differs then compiler will generate error

Pointers

- A *pointer* is a variable that contains a memory address. Very often this address is the location of another variable. For example, if **x** contains the address of **y**, then **x** is said to "point to" **y**.
- two special operators that are used with pointers: * and **&**.
- Pointer variables must be declared as such. The general form of a pointer variable declaration is

*type *var-name*;

```
#include <iostream>
using namespace std;
int main()
{
    int* p;
    double *q;
    return 0;
}
```

- While declaration, use asterisk between data type and variable name
- Asterik can stick to wither data type or variable name (both are fine)

Declaration and definition

```
#include <iostream>
using namespace std;
int main()
{
    int* p;
    int x;

    double *q;
    double y;

    p = &x;
    q = &y;

    return 0;
}
```

Declaration

Definition

p = & x ;

pointer

Ampersand sign

Some variable

example

```
#include <iostream>
using namespace std;
int main()
{
    int* p;
    int x;
    double *q;
    double y;
    p = &x;
    q = &y;
    cout << "Value of variables:" << endl;
    cout << "p:" << p << endl;
    cout << "x:" << x << endl;
    cout << "q:" << q << endl;
    cout << "y:" << y << endl;
    cout << "Address of variables:" << endl
    ;
    cout << "p:" << &p << endl;
    cout << "x:" << &x << endl;
    cout << "q:" << &q << endl;
    cout << "y:" << &y << endl;
      return 0;
}
```

Microsoft Visual Studio Debug Console

```
Value of variables:
p:012FFDA4
x:-858993460
q:012FFD88
y:-9.25596e+61
Address of variables:
p:012FFDB0
x:012FFDA4
q:012FFD98
y:012FFD88
```

Pointer **p** holding the address of variable **x**

Programming fundamentals

Lecture 18: *Pointers, pass by pointer, dereferencing a pointer*

Recap

- Passing 2D arrays
- Static variables
- Function overloading
- Introduction to pointers

1D array and functions

arr

1	2	3	4	-858993460	-1178649665	14220860	-1228649665	-778649665
0FXX0								
0	1	2	3	4	5	6	7	8

```
#include <iostream>
using namespace std;
void print(int arr[])
{
    for (int i=0;i<7;i++)
    {
        cout << arr[i] << " " << endl;
    }
}
int main()
{
    int arr[4] = {1,2,3,4};
    print(arr);
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
1
2
3
4
-858993460
-1178649665
14220860
```

2D array

```
#include <iostream>
using namespace std;
void print(int arr[][3])
{
    for (int i = 0; i < 2; i++)
    {
        for (int j=0 ; j<3 ; j++)
        {
            cout << "Value arr[" << i <<
            ","<<j<< "]: " << arr[i][j]
            << " " << endl;
            cout << "Address arr[" << i
            << "," << j << "]: " << &arr
            [i][j] << " " << endl;
        }
    }
}
int main()
{
    int arr[2][3] = {1,2,3,4,5,6};
    print(arr);
    return 0;
}
```

0,0	0,1	0,2
007CFE5C	007CFE60	007CFE64
1	*2*	*3*
1,0	1,1	1,2
007CFE68	007CFE6C	007CFE70
4	*5*	*6*

0	007CFE5C	1
1	007CFE60	2
2	007CFE64	3
3	007CFE68	4
4	007CFE6C	5
5	007CFE70	6
6	007CFE74	-858993460
7	007CFE78	1636287196
8	007CFE7C	*8191644*
9	007CFE80	*3485315*
10	007CFE84	1
11	007CFE88	*12214168*
12	007CFE8C	*12232232*
13	007CFE90	1
14	007CFE94	*12214168*
15	……	

```
#include <iostream>
using namespace std;
void print(int arr[][3])
{
    for (int i = 0; i < 5; i++)
    {
        for (int j=0 ; j<5 ; j++)
        {
            cout << "Value arr[" << i <<","<<j<< "]: " <<
            arr[i][j] << " " << endl;
            cout << "Address arr[" << i << "," << j << "]: "
            << &arr[i][j] << " " << endl;
        }
    }
}
int main()
{
    int arr[2][3] = {1,2,3,4,5,6};
    print(arr);
    return 0;
}
```

0	007CFE5C	1
1	007CFE60	2
2	007CFE64	3
3	007CFE68	4
4	007CFE6C	5
5	007CFE70	6
6	007CFE74	-858993460
7	007CFE78	1636287196
8	007CFE7C	8191644
9	007CFE80	3485315
10	007CFE84	1
11	007CFE88	12214168
12	007CFE8C	12232232
13	007CFE90	1
14	007CFE94	12214168
15	……	

0,0	0,1	0,2	0,3	0,4
007CFE5C	007CFE60	007CFE64	007CFE68	007CFE6C
1	2	3	4	5
1,0	1,1	1,2	1,3	1,4
007CFE68	007CFE6C	007CFE70	007CFE74	007CFE78
4	5	6	-858993460	1636287196
2,0	2,1	2,2	2,3	2,4
007CFE74	007CFE78	007CFE7C	007CFE80	007CFE84
-858993460	1636287196	8191644	3485315	1
3,0	3,1	3,2	3,3	3,4
007CFE80	007CFE84	007CFE88	007CFE8C	007CFE90
3485315	1	12214168	12232232	1
4,0	4,1	4,2	4,3	4,4
007CFE8C	007CFE90	007CFE94	007CFE8C	007CFE90
12232232	1	12214168	12232232	1

Agenda

- Using Pointers
- Dereferencing a pointer
- Pass by pointer

Pointers

- A *pointer* is a variable that contains a memory address. Very often this address is the location of another variable. For example, if **x** contains the address of **y**, then **x** is said to "point to" **y**.
- two special operators that are used with pointers: * and **&**.
- Pointer variables must be declared as such. The general form of a pointer variable declaration is

*type *var-name*;

```
#include <iostream>
using namespace std;
int main()
{
    int* p;
    double *q;
    return 0;
}
```

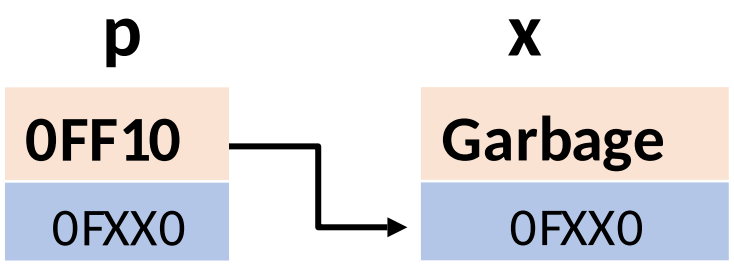
- While declaration, use asterisk between data type and variable name
- Asterik can stick to wither data type or variable name (both are fine)

Declaration and definition

```
#include <iostream>
using namespace std;
int main()
{
    int* p;
    int x;

    double *q;
    double y;

    p = &x;
    q = &y;

    return 0;
}
```

Declaration

Definition



p = & x ;

pointer

Ampersand sign

Some variable

example

```
#include <iostream>
using namespace std;
int main()
{
    int* p;
    int x;
    double *q;
    double y;
    p = &x;
    q = &y;
    cout << "Value of variables:" << endl;
    cout << "p:" << p << endl;
    cout << "x:" << x << endl;
    cout << "q:" << q << endl;
    cout << "y:" << y << endl;
    cout << "Address of variables:" << endl
    ;
    cout << "p:" << &p << endl;
    cout << "x:" << &x << endl;
    cout << "q:" << &q << endl;
    cout << "y:" << &y << endl;
      return 0;
}
```

The addresses you see when you run the program may differ from those shown here, but the net effect will be the same

Microsoft Visual Studio Debug Console

```
Value of variables:
p:012FFDA4
x:-858993460
q:012FFD88
y:-9.25596e+61
Address of variables:
p:012FFDB0
x:012FFDA4
q:012FFD98
y:012FFD88
```

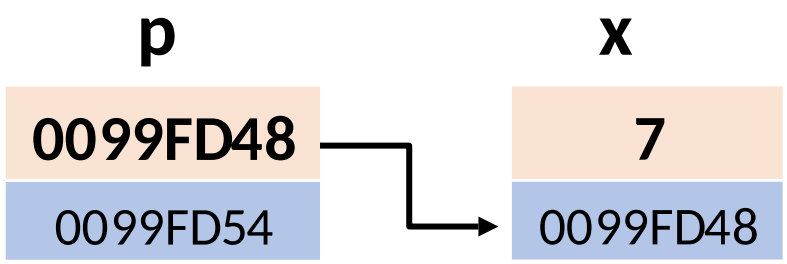
Pointer **p** holding the address of variable **x**

* operator

- Remember we used * while declaring pointer
- We can also use * to extract/fetch the value pointed by the pointer. (***also known as dereferencing***)

```
#include <iostream>
using namespace std;
int main()
{
int* p; int x = 7;
p = &x;
cout << "Value present at the location pointed by p:" << *p <<
endl;
return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int* p; int x = 7;
    cout << "Value of x :" << x << endl;
    cout << "Address of x :" << &x << endl;
    p = &x;
    cout << "Value of p :" << p << endl;
    cout << "Address of p :" << &p << endl;
    cout << "Value present at the location pointed by p :" << *p << endl
    ;
    return 0;
}
```



Microsoft Visual Studio Debug Console

```
Value of x:7
Address of x:0099FD48
Value of p:0099FD48
Address of p:0099FD54
Value present at the location pointed by p:7
```

Assigning Values Through a Pointer

```
#include <iostream>
using namespace std;
int main()
{
   int* p; int x = 7;
   int* q; int y = 9;
   //while assigning value, make sure L.H.S matches with R.H.S
   p = &x;
   q = &y;
   int* r, *s;
   r = p;
   s = q;
   return 0;
}
```

p = &x;

Pointer/address address

r = p;

Pointer/address Pointer/address

p	q	x	y
0FXY0	0FXY9	7	9
0FXX0	0FXX4	0FXY0	0FXY9

r	s	p	q	x	y
0FXY0	0FXY9	0FXY0	0FXY9	7	9
0FFF0	0FFF4	0FXX0	0FXX4	0FXY0	0FXY9

Change the value of variables indirectly using pointers

```
#include <iostream>
using namespace std;
int main()
{
   int* p; int x = 7;
   int* q; int y = 9;
   p = &x;
   q = &y;
   *p = 17;
   *q = *q + 10;
   return 0;
}
```

*p = 17;

value value

*q = *q+10;

value value

p	q	x	y
0FXY0	0FXY9	7	9
0FXX0	0FXX4	0FXY0	0FXY9

p	q	x	y
0FXY0	0FXY9	17	19
0FXX0	0FXX4	0FXY0	0FXY9

Incrementing pointer vs incrementing value pointed by a pointer

```
#include <iostream>
using namespace std;
int main()
{
    int* p; int x = 7;
    p = &x;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    (*p)++;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    p++;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int* p; int x = 7;
    p = &x;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    (*p)++;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    p++;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
p:00CFFD00
*p:7
&p:00CFFD0C
&x:00CFFD00
p:00CFFD00
*p:8
&p:00CFFD0C
&x:00CFFD00
p:00CFFD04
*p:-858993460
&p:00CFFD0C
&x:00CFFD00
```

x	**00CFFD00**	**7**
	00CFFD04	-858993460
p	00CFFD0C	**00CFFD00**

(*p)++ will indirectly print the value!

p++ will increment in the contents of p (that is an address), therefore immediate next location (to variable x) address gets stored in p.

Since its an int pointer therefore 4 bytes difference is there between consecutive locations.

Print the address of 5 memory locations, immediate next to x

```
#include <iostream>
using namespace std;
int main()
{
   int* p; int x = 7;
   p = &x;
   cout << "&x :" << &x << endl;
   for (int i = 0; i < 5; i++)
      cout << "Address# " << i <<
      " :" << p++ << endl;
   return 0;
}
```

Microsoft Visual Studio Debug Console

```
&x:006FF914
Address# 0 :006FF914
Address# 1 :006FF918
Address# 2 :006FF91C
Address# 3 :006FF920
Address# 4 :006FF924
```

Print the address of 5 memory locations, immediate next to x and also print the values present at these locations

```
#include <iostream>
using namespace std;
int main()
{
    int* p; int x = 7;
    p = &x;
    cout << "&x :" << &x << endl;
    for (int i = 0; i < 5; i++)
    {
        cout << "Address# " << i << " :" << p++ <<
        endl;
        cout << "Value# " << i << " :" << *p << endl;
    }
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
&x:0073FB44
Address# 0 :0073FB44
Value# 0 :-858993460
Address# 1 :0073FB48
Value# 1 :-858993460
Address# 2 :0073FB4C
Value# 2 :7600976
Address# 3 :0073FB50
Value# 3 :-858993460
Address# 4 :0073FB54
Value# 4 :1957166254
```

Pointers and Arrays

```
#include <iostream>
using namespace std;
int main()
{
    int arr[4] = {1,2,3,4};
    int* p;
    p = arr;
    cout << "p: " << p << endl;
    cout << "*p: " << *p << endl;
    cout << "&p: " << &p << endl;
    cout << "str: " << arr << endl;
    cout << "str[0]: " << arr[0] << endl;
    cout << "&str: " << &arr << endl;
    cout << "&str[0]: " << &arr[0] << endl;
    return 0;
}
```

Since array's name depicts base address therefore we can assign it to a pointer.
Provided that array type and pointer type matches

Microsoft Visual Studio Debug Console

```
p: 004FF9E0
*p: 1
&p: 004FF9D4
str: 004FF9E0
str[0]: 1
&str: 004FF9E0
&str[0]: 004FF9E0
```

Regarding atomic functions and good programming practice

Pls review the following code which is logically and syntactically fine but functions are not atomic. Lets explore the functions one by one and correct them..

```
#include<iostream>
using namespace std;
int hour, minute, second;
int input(int, int, int);
int addition(int, int, int);
int charges(int total_hours, int total_minutes, int total_seconds);
int main()
{
	input(0, 0, 0);
}
int input(int, int, int)
{
	int total_hours = 0, total_minutes = 0, total_seconds = 0;

	cout << "Enter the time the internet was used for in each sitting(in hours,
minutes and seconds).\nEnter -1 to end.\nEnter time of First Sitting : ";
	cout << "\nHours: ";
	cin >> hour;
	do
	{
		total_hours = total_hours + hour;
		cout << "Minute: ";
		cin >> minute;
		total_minutes = total_minutes + minute;
		cout << "Seconds: ";
		cin >> second;
		total_seconds = total_seconds + second;
		cout << "Enter time of next Sitting : ";
		cout << "\nHours: ";
		cin >> hour;
	} while (hour != -1);
	addition(total_hours, total_minutes, total_seconds);
	return 0;
}
int addition(int total_hours, int total_minutes, int total_seconds)
{
	total_minutes = total_minutes + (total_seconds / 60);
	total_hours = total_hours + (total_minutes / 60);
	charges(total_hours, total_minutes, total_seconds);
	return 0;
}
int charges(int total_hours, int total_minutes, int total_seconds)
{
	cout << "Total time of usage is " << total_hours << endl;
	cout << "Total charges are Rs. " << total_hours * 25;
	return 0;
}
```

Input function:

This function should logically help in taking input from the user. The input should obviously must be stored in variables and this function should not return anything because this is not required!

Input/Output: technically this function does not need any thing from us (the user) to take input. And it also does not need to return anything. E.g. in case of FindSum function we need a **sum** from this function. So, ideally the function should look like:

```
void input()
{
        //declare relevant variables
        //and take input in then through cin
}
```

But, we also know that we need the values that you take from the user (hours, minutes, seconds) in other functions (add to total and calculate charges). Thus, we need to do some adjustments and ordering in our code lines to make these values available to the other functions. There are three possibilities for it:

1) **Declare hours, minutes, seconds as global variables**

   But, we already studied that we should not use global variables extensively in our code. We need it in scenarios when we have to deal with constants that are technically common to whole application. .e.g interest rate in bank application. We cannot declare our regular used variables (used in our logic design) as global variables.

2) **Call the other function that need input values in the input function itself like:**

```
void input()
{
        //declare relevant variables
        //and take input in then through cin
        //call the function(s)
        addition(total_hours, total_minutes, total_seconds);

}
```

   This is also not a good programming practice. The reason is that input function must only consists of those lines of code/ logic that is related to input only. It should not contain any other functionality, like in this case we are adding function call to achieve addition.

3) **Use reference variables**

   The third and acceptable option is to use reference variables. Pass reference of variables to function so that the input function takes input in original variables (that are referred).

```
int main()
{
        input(0, 0, 0);
}
int input(int, int, int)
{
        int total_hours = 0, total_minutes = 0, total_seconds = 0;

        cout << "Enter the time the internet was used for in each sitting(in hours, minutes and
seconds).\nEnter -1 to end.\nEnter time of First Sitting : ";
        cout << "\nHours: ";
        cin >> hour;
        do
        {
                total_hours = total_hours + hour;
                cout << "Minute: ";
                cin >> minute;
                total_minutes = total_minutes + minute;
                cout << "Seconds: ";
                cin >> second;
                total_seconds = total_seconds + second;
                cout << "Enter time of next Sitting : ";
                cout << "\nHours: ";
                cin >> hour;
        } while (hour != -1);
        addition(total_hours, total_minutes, total_seconds);
        return 0;
}
```

Thus better implementation of this function would be:

```
void input(int &hour,int &minute, int &second)
{
              cout << "Enter the time the internet was used for in each
sitting(in hours, minutes and seconds).\nEnter -1 to end.\nEnter time of First
Sitting : ";
              cout << "\nHours: ";
              cin >> hour;
              cout << "Minute: ";
              cin >> minute;
              cout << "Seconds: ";
              cin >> second;
}
```

addtoTotal/addition function:

This function should logically help in adding the inputs taken from the user to some variable. That is, hours, minutes, and seconds for each sitting must add up to total_hours, total_minutes, and total_seconds, respectively. Lets explore the following function.

```
int addition(int total_hours, int total_minutes, int total_seconds)
{
       total_minutes = total_minutes + (total_seconds / 60);
       total_hours = total_hours + (total_minutes / 60);
       charges(total_hours, total_minutes, total_seconds);
       return 0;
}
```

Input/Output: technically this function does not need anything from us (the user) to add up the inputs gradually. However, it do need from us the values of input hours/minutes/seconds because that needs to be added up for each sitting. With the same motivation of reference variables used for input function, we can use the same concept here and remove the call to **charges** function that is irrelevant for the scope of this function. The addition function should only contain functionality related to addition. Nothing more and nothing less!. Thus, a better version would be:

```
void addition(int &total_hours, int &total_minutes, int &total_seconds, int
hour, int second, int minute)
{
       total_hours = total_hours + hour;
       total_seconds = total_seconds + second;
       total_minutes = total_minutes + minute;
}
```

Calculate charges function:

This function should logically help in calculating the total charges. While calculating the charges the hours related calculation also needs to be performed (i.e. to convert minutes and seconds to hours). Any kind of conversion related logic should be the part of this function. You can think about inserting the **convert hours,minutes,seconds to HOURS** logic in addition function. But the purpose of addition function was to take hours one by one and add then to TOTAL HOURS. Same goes for minutes and seconds. Converting minutes to hours or seconds to minutes does not as such comes under its scope. So it is better to put that logic in charges function.

```
int charges(int total_hours, int total_minutes, int total_seconds)
{
	cout << "Total time of usage is " << total_hours << endl;
	cout << "Total charges are Rs. " << total_hours * 25;
	return 0;
}
```

A better implementation would be:

```
int charges(int total_hours, int total_minutes, int total_seconds)
{
	total_minutes = total_minutes + (total_seconds / 60);
	total_hours = total_hours + (total_minutes / 60);
	cout << "Total time of usage is " << total_hours << endl;
	cout << "Total charges are Rs. " << total_hours * 25;
	return 0;
}
```

You can have another print function that prints the final answer. But it is needless to do so as we discussed that too many smaller functions are actually an overhead to our code's performance. Therefore skip making too simple and too short multiple functions.

Now lets have a look at the main function (older and corrected version). The red version has only one call that meaninglessly takes three 0s as an input (Not required logically!).

```
int main()
{
	input(0, 0, 0);
}
```

Think of your main function as an interface. That is, the starting point of your application. Assume you start any game, lets say StreetFighter.. First, always an interface screen appears that takes input from you how you want to use the game application. You give input by pressing any button (settings, help, vs battle, team battle, quit). The main function then calls the relevant methods. Thus, ideally your main should only contain calls to functions. Yes, it CAN contain few other statements too, like variable declarations, or some messages to print anything on console.e.g welcome message. But you should try to keep it free from any manipulation or logic. Thus, a better implementation of main function would be:

```
int main()
{
	int hour=0, minute=0, second=0;
	int total_hours = 0, total_minutes = 0, total_seconds = 0;
	do
	{
		input(hour,minute,second);
		if(hour!=-1)
			addition(total_hours, total_minutes, total_seconds,
		hour,minute,second);
	} while (hour != -1);
	charges(total_hours, total_minutes, total_seconds);
}
```

To Summarize pls see full code versions (old + improved version)

```
#include<iostream>
using namespace std;
int hour, minute, second;
int input(int, int, int);
int addition(int, int, int);
int charges(int total_hours, int total_minutes, int total_seconds);
int main()
{
	input(0, 0, 0);
}
int input(int, int, int)
{
	int total_hours = 0, total_minutes = 0, total_seconds = 0;

	cout << "Enter the time the internet was used for in each sitting(in hours,
minutes and seconds).\nEnter -1 to end.\nEnter time of First Sitting : ";
	cout << "\nHours: ";
	cin >> hour;
	do
	{
		total_hours = total_hours + hour;
		cout << "Minute: ";
		cin >> minute;
		total_minutes = total_minutes + minute;
		cout << "Seconds: ";
		cin >> second;
		total_seconds = total_seconds + second;
		cout << "Enter time of next Sitting : ";
		cout << "\nHours: ";
		cin >> hour;
	} while (hour != -1);
	addition(total_hours, total_minutes, total_seconds);
	return 0;
}
int addition(int total_hours, int total_minutes, int total_seconds)
{
	total_minutes = total_minutes + (total_seconds / 60);
	total_hours = total_hours + (total_minutes / 60);
	charges(total_hours, total_minutes, total_seconds);
	return 0;
}
int charges(int total_hours, int total_minutes, int total_seconds)
{
	cout << "Total time of usage is " << total_hours << endl;
	cout << "Total charges are Rs. " << total_hours * 25;
	return 0;
}
```

```
#include<iostream>
using namespace std;
void input(int &, int &,int &);
void addition(int&, int&, int&,int,int,int);
int charges(int total_hours, int total_minutes, int total_seconds);
int main()
{
	int hour=0, minute=0, second=0;
	int total_hours = 0, total_minutes = 0, total_seconds = 0;
	do
	{
		input(hour,minute,second);
		if(hour!=-1)
			addition(total_hours, total_minutes, total_seconds,
hour,minute,second);
	} while (hour != -1);
	charges(total_hours, total_minutes, total_seconds);
}
void input(int &hour,int &minute, int &second)
{
		cout << "Enter the time the internet was used for in each sitting(in
hours, minutes and seconds).\nEnter -1 to end.\nEnter time of First Sitting : ";
		cout << "\nHours: ";
		cin >> hour;
		cout << "Minute: ";
		cin >> minute;
		cout << "Seconds: ";
		cin >> second;
}
void addition(int &total_hours, int &total_minutes, int &total_seconds, int hour,
int second, int minute)
{
	total_hours = total_hours + hour;
	total_seconds = total_seconds + second;
	total_minutes = total_minutes + minute;
}
int charges(int total_hours, int total_minutes, int total_seconds)
{
	total_minutes = total_minutes + (total_seconds / 60);
	total_hours = total_hours + (total_minutes / 60);
	cout << "Total time of usage is " << total_hours << endl;
	cout << "Total charges are Rs. " << total_hours * 25;
	return 0;
}
```

Programming fundamentals

Lecture 19: *Pointers and arrays*

Recap

- Using Pointers
- Dereferencing a pointer

Online lectures log

- https://docs.google.com/spreadsheets/d/1LoI1nyp5jRlI-H1FTRFK8JS7saoNvNimhpueWdZFsMo/edit?usp=sharing

example

```
#include <iostream>
using namespace std;
int main()
{
    int* p;
    int x;
    double *q;
    double y;
    p = &x;
    q = &y;
    cout << "Value of variables:" << endl;
    cout << "p:" << p << endl;
    cout << "x:" << x << endl;
    cout << "q:" << q << endl;
    cout << "y:" << y << endl;
    cout << "Address of variables:" << endl
    ;
    cout << "p:" << &p << endl;
    cout << "x:" << &x << endl;
    cout << "q:" << &q << endl;
    cout << "y:" << &y << endl;
      return 0;
}
```

The addresses you see when you run the program may differ from those shown here, but the net effect will be the same

Microsoft Visual Studio Debug Console

```
Value of variables:
p:012FFDA4
x:-858993460
q:012FFD88
y:-9.25596e+61
Address of variables:
p:012FFDB0
x:012FFDA4
q:012FFD98
y:012FFD88
```

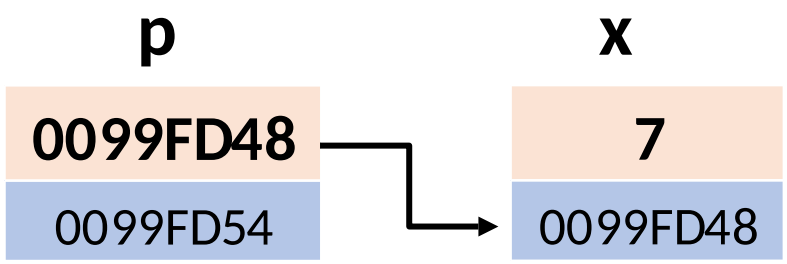
Pointer **p** holding the address of variable **x**

* operator

- Remember we used * while declaring pointer
- We can also use * to extract/fetch the value pointed by the pointer. (***also known as dereferencing***)

```
#include <iostream>
using namespace std;
int main()
{
int* p; int x = 7;
p = &x;
cout << "Value present at the location pointed by p:" << *p <<
endl;
return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int* p; int x = 7;
    cout << "Value of x :" << x << endl;
    cout << "Address of x :" << &x << endl;
    p = &x;
    cout << "Value of p :" << p << endl;
    cout << "Address of p :" << &p << endl;
    cout << "Value present at the location pointed by p :" << *p << endl
    ;
    return 0;
}
```



Microsoft Visual Studio Debug Console

```
Value of x:7
Address of x:0099FD48
Value of p:0099FD48
Address of p:0099FD54
Value present at the location pointed by p:7
```

Assigning Values Through a Pointer

```
#include <iostream>
using namespace std;
int main()
{
   int* p; int x = 7;
   int* q; int y = 9;
   //while assigning value, make sure L.H.S matches with R.H.S
   p = &x;
   q = &y;
   int* r, *s;
   r = p;
   s = q;
   return 0;
}
```

p = &x;

Pointer/address address

r = p;

Pointer/address Pointer/address

p	q	x	y
0FXY0	0FXY9	7	9
0FXX0	0FXX4	0FXY0	0FXY9

r	s	p	q	x	y
0FXY0	0FXY9	0FXY0	0FXY9	7	9
0FFF0	0FFF4	0FXX0	0FXX4	0FXY0	0FXY9

Change the value of variables indirectly using pointers

```
#include <iostream>
using namespace std;
int main()
{
   int* p; int x = 7;
   int* q; int y = 9;
   p = &x;
   q = &y;
   *p = 17;
   *q = *q + 10;
   return 0;
}
```

*p = 17;

value value

*q = *q+10;

value value

p	q	x	y
0FXY0	0FXY9	7	9
0FXX0	0FXX4	0FXY0	0FXY9

p	q	x	y
0FXY0	0FXY9	17	19
0FXX0	0FXX4	0FXY0	0FXY9

Incrementing pointer vs incrementing value pointed by a pointer

```
#include <iostream>
using namespace std;
int main()
{
    int* p; int x = 7;
    p = &x;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    (*p)++;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    p++;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int* p; int x = 7;
    p = &x;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    (*p)++;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    p++;
    cout << "p :" << p << endl;
    cout << "*p :" << *p << endl;
    cout << "&p :" << &p << endl;
    cout << "&x :" << &x << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
p:00CFFD00
*p:7
&p:00CFFD0C
&x:00CFFD00
p:00CFFD00
*p:8
&p:00CFFD0C
&x:00CFFD00
p:00CFFD04
*p:-858993460
&p:00CFFD0C
&x:00CFFD00
```

x	**00CFFD00**	**7**
	00CFFD04	-858993460
p	00CFFD0C	**00CFFD00**

(*p)++ will indirectly print the value!

p++ will increment in the contents of p (that is an address), therefore immediate next location (to variable x) address gets stored in p.

Since its an int pointer therefore 4 bytes difference is there between consecutive locations.

Agenda

- Pointers and arrays
- Array of pointers

Print the address of 5 memory locations, immediate next to x

```
#include <iostream>
using namespace std;
int main()
{
   int* p; int x = 7;
   p = &x;
   cout << "&x :" << &x << endl;
   for (int i = 0; i < 5; i++)
      cout << "Address# " << i <<
      " :" << p++ << endl;
   return 0;
}
```

Microsoft Visual Studio Debug Console

```
&x:006FF914
Address# 0 :006FF914
Address# 1 :006FF918
Address# 2 :006FF91C
Address# 3 :006FF920
Address# 4 :006FF924
```

Print the address of 5 memory locations, immediate next to x and also print the values present at these locations

```
#include <iostream>
using namespace std;
int main()
{
    int* p; int x = 7;
    p = &x;
    cout << "&x :" << &x << endl;
    for (int i = 0; i < 5; i++)
    {
        cout << "Address# " << i << " :" << p++ <<
        endl;
        cout << "Value# " << i << " :" << *p << endl;
    }
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
&x:0073FB44
Address# 0 :0073FB44
Value# 0 :-858993460
Address# 1 :0073FB48
Value# 1 :-858993460
Address# 2 :0073FB4C
Value# 2 :7600976
Address# 3 :0073FB50
Value# 3 :-858993460
Address# 4 :0073FB54
Value# 4 :1957166254
```

Pointers and Arrays

Since array's name depicts base address therefore we can assign it to a pointer.
Provided that array type and pointer type matches

```
#include <iostream>
using namespace std;
int main()
{
    int arr[4] = {1,2,3,4};
    int* p;
    p = arr;
    cout << "p: " << p << endl;
    cout << "*p: " << *p << endl;
    cout << "&p: " << &p << endl;
    cout << "str: " << arr << endl;
    cout << "str[0]: " << arr[0] << endl;
    cout << "&str: " << &arr << endl;
    cout << "&str[0]: " << &arr[0] << endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
p: 004FF9E0
*p: 1
&p: 004FF9D4
str: 004FF9E0
str[0]: 1
&str: 004FF9E0
&str[0]: 004FF9E0
```

Accessing any element of an array with a pointer

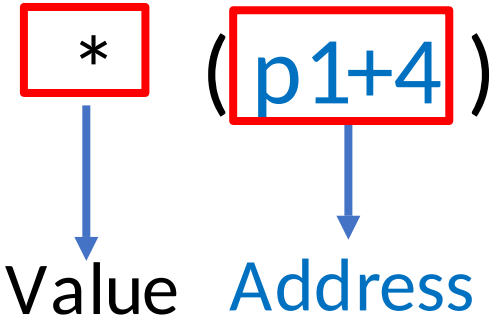
```
#include <iostream>
using namespace std;
int main()
{
   char str[80];
   char* p1;
   p1 = str;
}
```

p1 = str assigns the address of str[0] to p1 Since, after the assignment, p1 points to the beginning of str, you may use p1 to access elements in the array. For example, if you want to access the fifth element in str, you could use

str[4]

or

*(p1+4)



The parentheses surrounding p1+4 are necessary because the * operation has a higher priority than the + operation.
Without the parentheses, the expression would first find the value pointed to by p1 (the first location in the array) and then add 4 to it.

*p1+4

↓

'a'+4

↓

98+4=101

The parentheses surrounding p1+4 are necessary because the * operation has a higher priority than the + operation. Without the parentheses, the expression would first find the value pointed to by p1 (the first location in the array) and then add 4 to it.

Iterating an array using pointer

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5]= {1,2,3,4,5};
    int* p;
    p = arr;
    for (int i = 0; i < 5; i++)
    {
        cout << *p << endl;
        p++;
    }
}
```

p++

Microsoft Visual Studio Debug Console

```
1
2
3
4
5
```

Iterating an array using pointer

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5]= {1,2,3,4,5};
    int* p;
    p = arr;
    for (int i = 0; i < 5; i++)
    {
        cout << *p++ << endl;
    }
}
```

*p++

Post increment
It will add first, ++ has highest precedence

Microsoft Visual Studio Debug Console

```
1
2
3
4
5
```

Are Pointers and Arrays Interchangeable?

- pointers and arrays are not completely interchangeable

```
#include <iostream>
using namespace std;
int main()
{
   int num[10]; int* p;
   int i;
   p = num;
   for (i = 0; i < 10; i++) {
      p++; // this is OK
      num++; // ERROR -- cannot modify num
   }
}
```

You cannot increment base address by simply using array name. But, you can increment pointer

```
#include <iostream>
using namespace std;
int main()
{
   int num[10]; int* p;
   int i;
   p = num;
   for (i = 0; i < 10; i++) {
      *num = i;// this is OK
      num++; // ERROR -- cannot modify num
   }
}
```

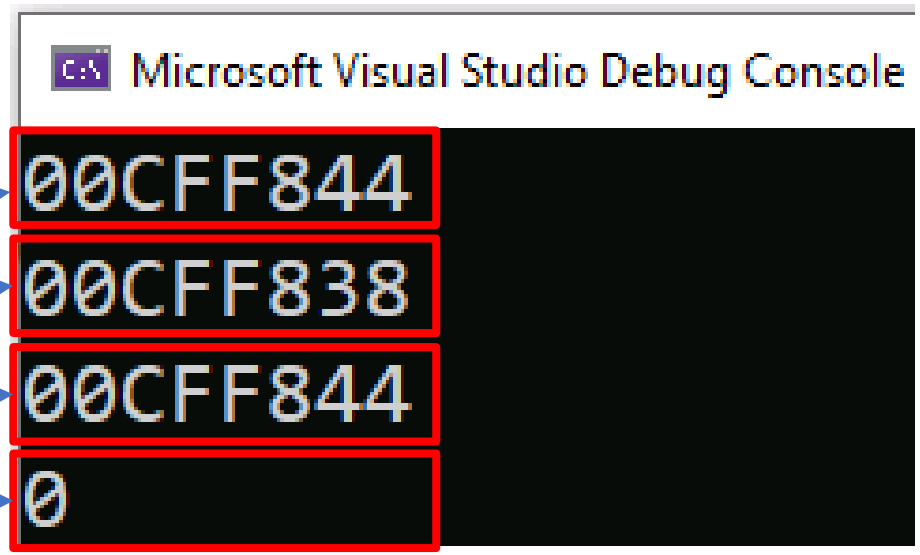
However, it is perfectly acceptable to apply the * operator to num
i.e. *num=I;

Arrays of Pointers

P[0]	P[1]	P[2]
00CFF838	00CFF888	00CFF788
00CFF844	00CFF848	00CFF852

x	y	z
0	1	2
00CFF838	00CFF888	00CFF788

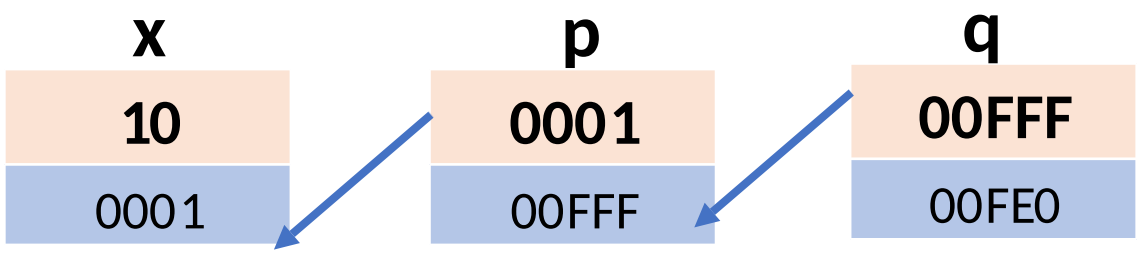
```
#include <iostream>
using namespace std;
int main()
{
    int* p[3];
    int x = 0, y = 1, z = 2;
    p[0] = &x;
    p[1] = &y;
    p[2] = &z;
    cout << p << endl;
    cout << p[0] << endl;
    cout << &p[0] << endl;
    cout << *p[0] << endl;
}
```



Multiple indirections

- A pointer to a pointer is a form of multiple indirection, or a chain of pointers.

```
#include <iostream>
using namespace std;
int main()
{
   int x, * p, ** q;
   x = 10;
   p = &x;
   q = &p;
   cout << *q; // prints the value of p

   cout << **q; // prints the value of x
   return 0;
}
```



0001

10

Pointer initialization is must!

- Just as we must make sure to initialize or define variables, the same applies to pointers

```
#include <iostream>
using namespace std;
int main()
{
    int x, * p;
    x = 10;
    *p = x; // where does p
    point?
    return 0;
}
```

ERROR

Programming fundamentals

Lecture 20: *Array of pointers*, Dynamic Allocation Using new and delete, *pass by pointers* , *functions and pointers*

Recap

- Pointers and arrays

Pointers and Arrays

```
#include <iostream>
using namespace std;
int main()
{
    int arr[4] = {1,2,3,4};
    int* p;
    p = arr;
    cout << "p: " << p << endl;
    cout << "*p: " << *p << endl;
    cout << "&p: " << &p << endl;
    cout << "str: " << arr << endl;
    cout << "str[0]: " << arr[0] << endl;
    cout << "&str: " << &arr << endl;
    cout << "&str[0]: " << &arr[0] << endl;
    return 0;
}
```

Since array's name depicts base address therefore we can assign it to a pointer.
Provided that array type and pointer type matches

Microsoft Visual Studio Debug Console

```
p: 004FF9E0
*p: 1
&p: 004FF9D4
str: 004FF9E0
str[0]: 1
&str: 004FF9E0
&str[0]: 004FF9E0
```

Accessing any element of an array with a pointer

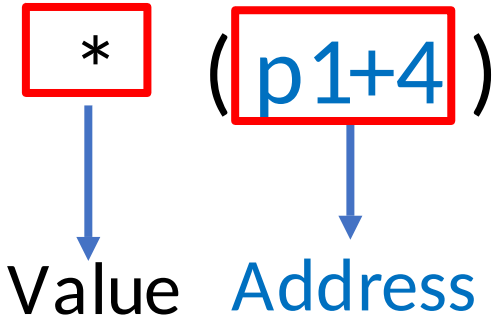
```
#include <iostream>
using namespace std;
int main()
{
   char str[80];
   char* p1;
   p1 = str;
}
```

p1 = str assigns the address of str[0] to p1 Since, after the assignment, p1 points to the beginning of str, you may use p1 to access elements in the array. For example, if you want to access the fifth element in str, you could use

str[4]

or

*(p1+4)



The parentheses surrounding p1+4 are necessary because the * operation has a higher priority than the + operation.
Without the parentheses, the expression would first find the value pointed to by p1 (the first location in the array) and then add 4 to it.

Are Pointers and Arrays Interchangeable?

- pointers and arrays are not completely interchangeable

```
#include <iostream>
using namespace std;
int main()
{
   int num[10]; int* p;
   int i;
   p = num;
   for (i = 0; i < 10; i++) {
      p++; // this is OK
      num++; // ERROR -- cannot modify num
   }
}
```

You cannot increment base address by simply using array name. But, you can increment pointer

Agenda

- Array of pointers
- Multiple indirections
- Initializing a pointer
- The Null Pointer Convention
- Dynamic Allocation Using new and delete
- Pass by pointer
- Using pointers in functions

Arrays of Pointers

P[0]	P[1]	P[2]
00CFF838	00CFF888	00CFF788
00CFF844	00CFF848	00CFF852

```
#include <iostream>
using namespace std;
int main()
{
    int* p[3];
    int x = 0, y = 1, z = 2;
    p[0] = &x;
    p[1] = &y;
    p[2] = &z;
    cout << p << endl;
    cout << p[0] << endl;
    cout << &p[0] << endl;
    cout << *p[0] << endl;
}
```

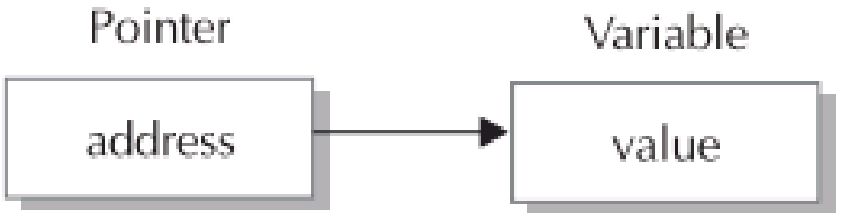
x	y	z
0	1	2
00CFF838	00CFF888	00CFF788

Microsoft Visual Studio Debug Console

```
00CFF844
00CFF838
00CFF844
0
```

Multiple indirections

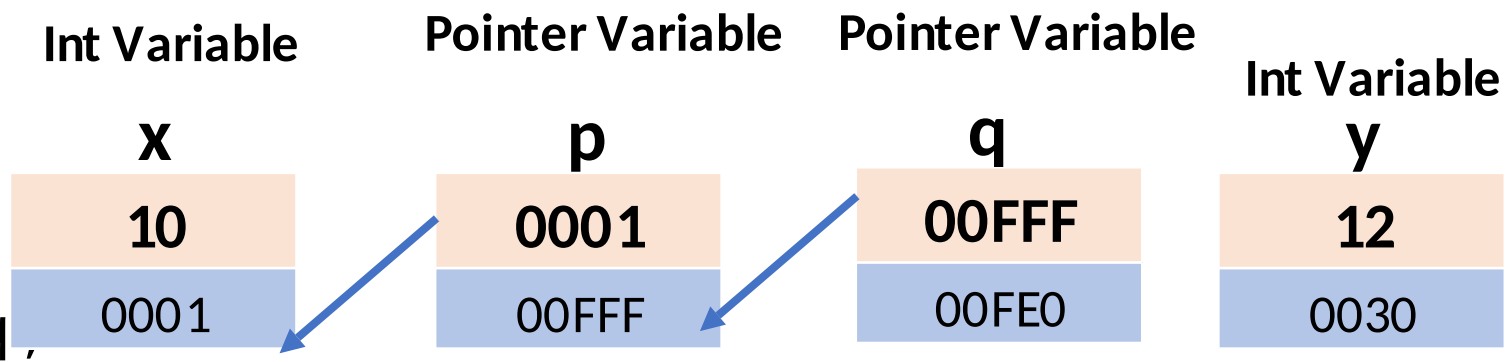
- A pointer to a pointer is a form of multiple indirection, or a chain of pointers.



Single Indirection

Multiple Indirection

```
#include <iostream>
using namespace std;
int main()
{
   int x, y, * p, ** q;
   x = 10; y=12;
   p = &x;
   q = &p;
   return 0;
}
```



p = &x — Valid

Pointer to int (address) variable = address of int

x = y — Valid

Int value = Int value

x = &y — Invalid

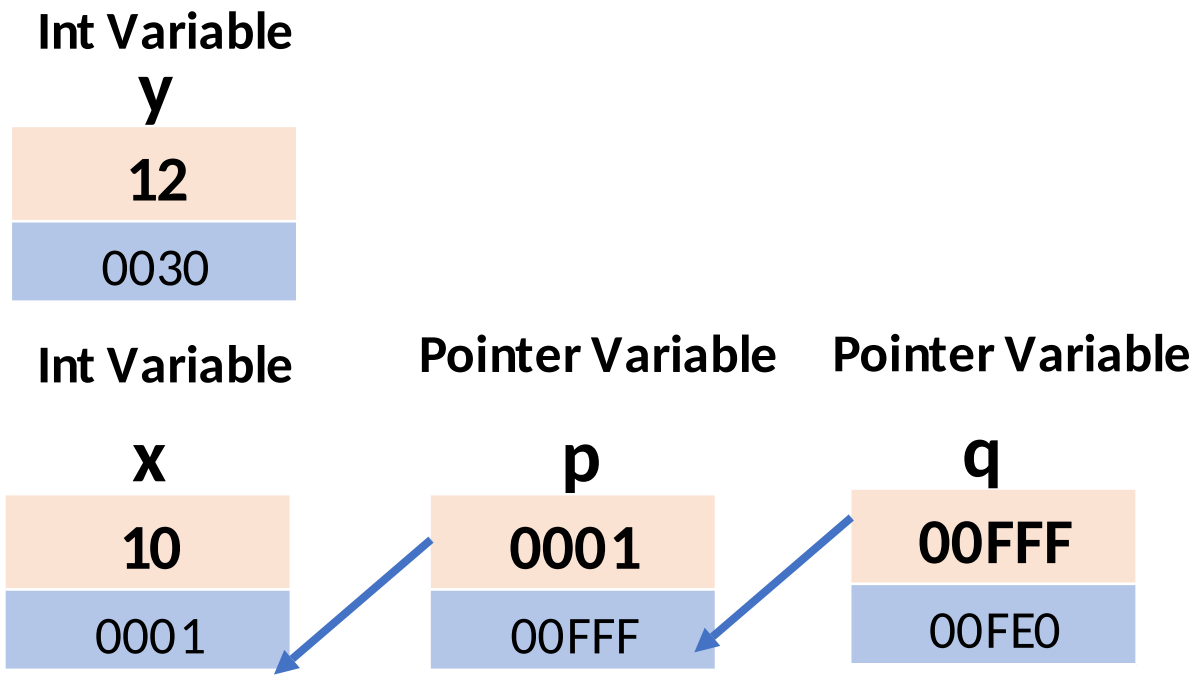
Int value = address of int variable

q = &x — Invalid

Pointer to pointer (address) = (address of int variable)

Multiple indirections

```
#include <iostream>
using namespace std;
int main()
{
   int x, y, * p, ** q;
   x = 10; y=12;
   p = &x;
   q = &p;
   cout << *q; // prints the value of p

   cout << **q; // prints the value of x
   return 0;
}
```



x = y

Int variable variable = Int

Valid

x = &y

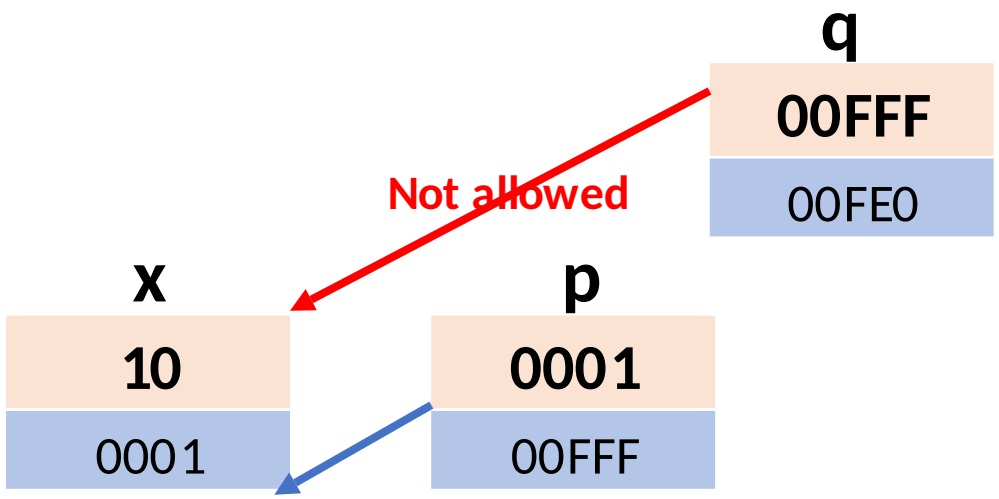
Int variable variable = Int

- We cannot assign * to **

```
#include <iostream>
using namespace std;
int main()
{
    int x, * p, ** q;
    x = 10;
    p = &x;
    q = &x;
    co
    cout << **q; // prints the value of x
    return 0;
}
```

a value of type "int *" cannot be assigned to an entity of type "int **"

Search Online



Pointer initialization is must!

- Just as we must make sure to initialize or define variables, the same applies to pointers

```
#include <iostream>
using namespace std;
int main()
{
    int x, *p;
    x = 10;
    *p = x; // where does p
    point?
    return 0;
}
```

ERROR

The Null Pointer Convention

- there is no sure way to avoid using an uninitialized pointer, C++ programmers have adopted a procedure that helps prevent some errors.
- By convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.
- Thus, if all unused pointers are given the null value and you avoid the use of a null pointer, you can avoid the accidental misuse of an uninitialized pointer.
- This is a good practice to follow.

```
float *p = 0; // p is now a null pointer
```

To check for a null pointer, use an if statement, like one of these:

```
if(p) // succeeds if p is not null
if(!p) // succeeds if p is null
```

Dynamic memory allocation

- There are two primary ways in which a C++ program can store information in the main memory of the computer.

1. **Static memory allocation** in which you cannot change memory at runtime. Memory is allocated at compiler time (**stack**)
2. **Dynamic memory allocation** in which memory is allocated at run time (**heap**)

- Dynamic memory allocated is possible with the help of pointers!
- Dynamic allocation is an important part of nearly all real-world programs
  - linked lists, graphs (used in modeling facebook like applications) and binary trees work on this concept

new and delete keyword

- C++ contains two operators, new and delete, that perform the functions of allocating and freeing memory.
- Their general forms are shown here:

```
pointer-var = new var-type;
delete pointer-var;
```

- Here, pointer-var is a pointer of type var-type. The new operator allocates sufficient memory to hold a value of type var-type and returns a pointer to it. Any valid data type can be allocated using new.
- The delete operator frees the memory pointed to by pointer-var. Once freed, this memory can be reallocated to different purposes by a subsequent new allocation request.

Dynamic memory allocation

```
#include <iostream>
using namespace std;
int main()
{
   int* p;
   p = new int; // allocate memory for int
   *p = 20; // assign that memory the value 20
   cout << *p; // prove that it works by displaying value
   delete p; // free the memory
   return 0;
}
```

What if we try to access memory after deleting it? Exception will be thrown!

```
#include <iostream>
using namespace std;
int main()
{
    int* p;
    p = new int; // allocate memory for int
    *p = 20; // assign that memory the value 20
    cout << *p; // prove that it works by displaying value
    delete p; // free the memory
    cout << *p;
    return 0;
}
```

Exception Thrown

Exception thrown: read access violation.

Initializing dynamically allocated memory

```
#include <iostream>
using namespace std;
int main()
{
   int* p;
   p = new int(99); // initialize with 99
   cout << *p; // displays 99
   delete p;
   return 0;
}
```

Allocating arrays

- Everything is same as we have done for allocating one memory location dynamically.
- Now we are going to explore how to allocate arrays
- Syntax for allocating and deallocating memory for arrays is:

```
pointer-var = new type [size];

delete [ ] pointer-var;
```

```
#include <iostream>
using namespace std;
int main()
{
   int* p;
   int i;
   p = new int[10]; // get a 10-element array
   for (i = 0; i < 10; i++)
     p[i] = i;
   delete[] p; // delete the entire array
   return 0;
}
```

	0	1	2	3	4	5	6	7	8	9
p	0000	0004	0008	0012	0016	0020	0024	0028	0032	0036

You can now have user defined size array with dynamic allocation

```
#include <iostream>
using namespace std;
int main()
{
   int* p;
   int size = 0;
   cout << "Enter the size:" << endl;
   cin >> size;
   p = new int[size]; // get a 10-element array
   for (int i = 0; i < size; i++)
   p[i] = i;
   delete[] p; // delete the entire array
   return 0;
}
```

Microsoft Visual Studio Debug Console

```
Enter the size:
5
```

Make sure to access only those memory that you allocated, otherwise your compiler will give exception in this case

Pointers in functions

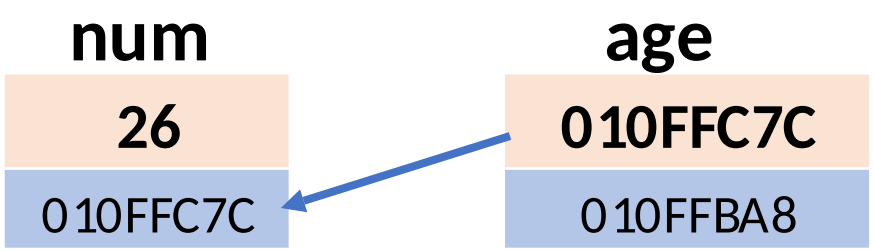
C++ allows you to pass a pointer to a function.

Pass by Pointe

Example 1:

pointer = variable's address *assignment*

```
#include <iostream>
using namespace std;
void printAge(int* age);

int main() {
    int num = 26;
    int* age;
    age = &num;
    return 0;
}
```



Example 1

```
#include <iostream>
using namespace std;
void func(int *age);

int main() {
    int num=26;
    cout << "num : " << num << endl;
    cout << "Address of num : " << &num <<
endl;
    func(&num);
    return 0;
}

void func(int *age) {
    cout << "Age received : " << age << endl;
    cout << "Address of age : " << &age <<
endl;
}
```

num	age
26	010FFC7C
010FFC7C	010FFBA8

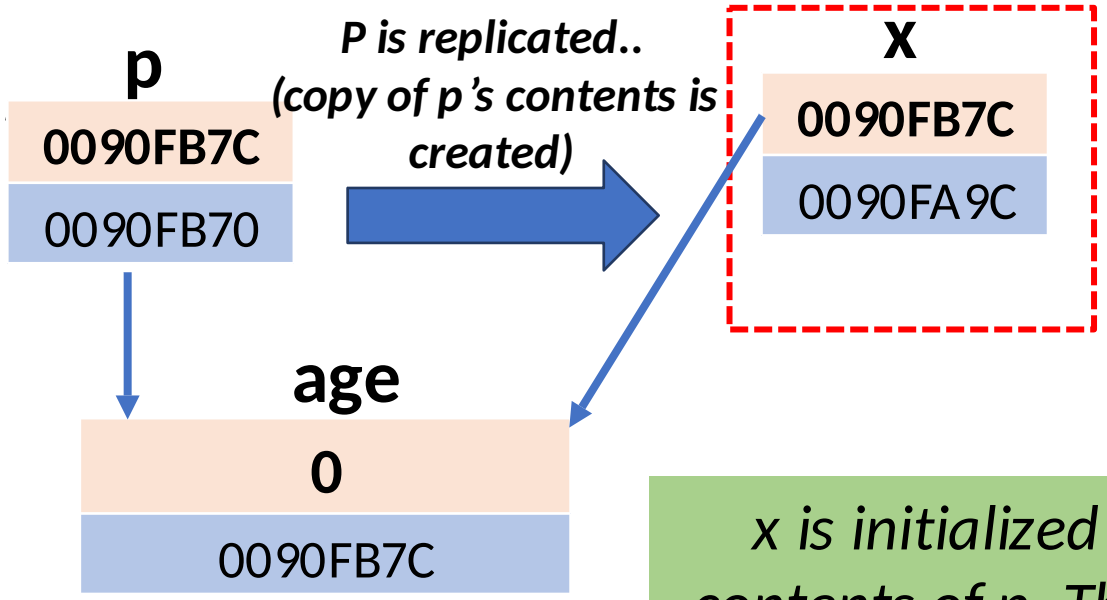
```
age = &num;
pointer = address;
```

Microsoft Visual Studio Debug Console

```
num: 26
Address of num: 010FFC7C
Age received: 010FFC7C
Address of age: 010FFBA8
```

Example 2:

pointer = pointer *assignmen*



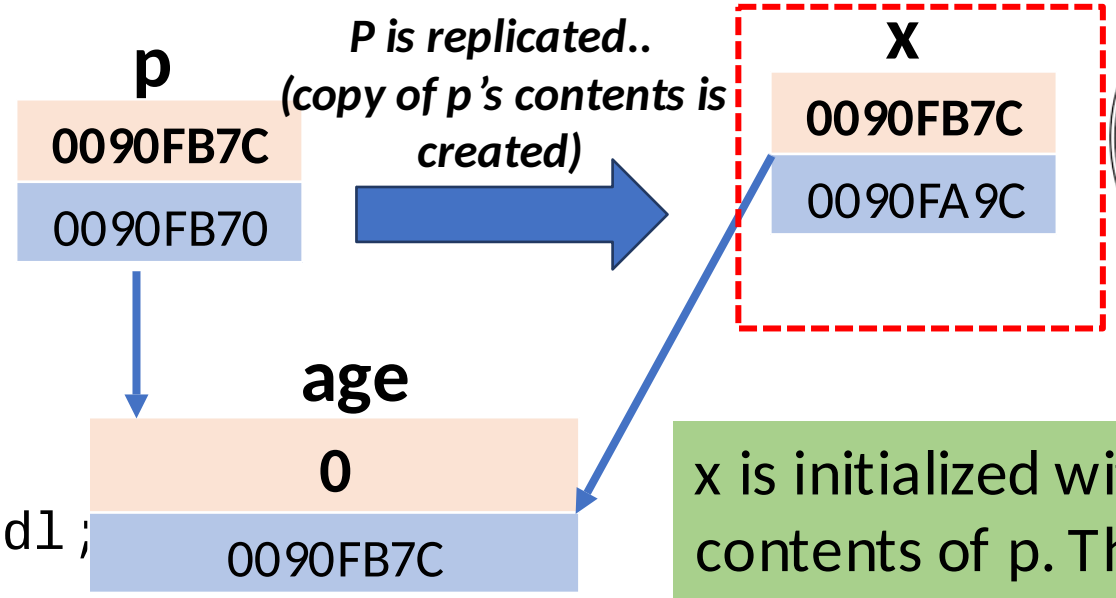

```
#include <iostream>
using namespace std;

int main()
{
    int age = 0; int* p; int* x;
    p = &age;
    x = p;
    return 0;
}
```

x is initialized with contents of p. That is, x also points where p was pointing. Because contents of pointer decide where to point!

```
#include <iostream>
using namespace std;
void printAge(int *x);

int main() {
    int age=0; int* p;
    p = &age;
    cout << "&age : " << &age << endl;
    cout << "p : " << p << endl;
    cout << "&p : " << &p << endl;
    printAge(p);
    return 0;
}

void printAge(int *x) {
    cout << "x : " << x << endl;
    cout << "&x : " << &x << endl;
}
```

x = p;

pointer = pointer;



x is initialized with contents of p. That is, x also points where p was pointing

Microsoft Visual Studio

```
&age: 0090FB7C
p: 0090FB7C
&p: 0090FB70
x: 0090FB7C
&x: 0090FA9C
```

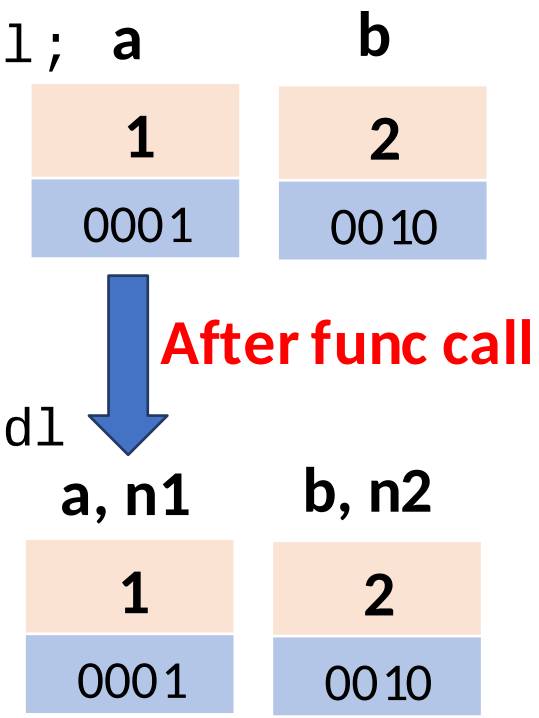
Pass by reference vs pass by pointers

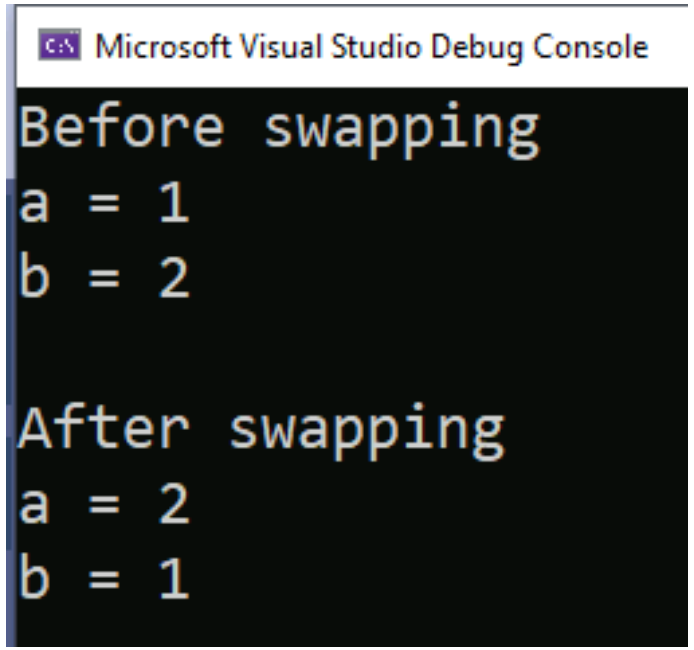
Comparison

Swapping code: pass by reference..

```
#include <iostream>
using namespace std;
void swap(int&, int&);
int main()
{
    int a = 1, b = 2;
    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    swap(a, b);

    cout << "\nAfter swapping" << endl
;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}
```

```
void swap(int &n1, int &n2) {
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
```



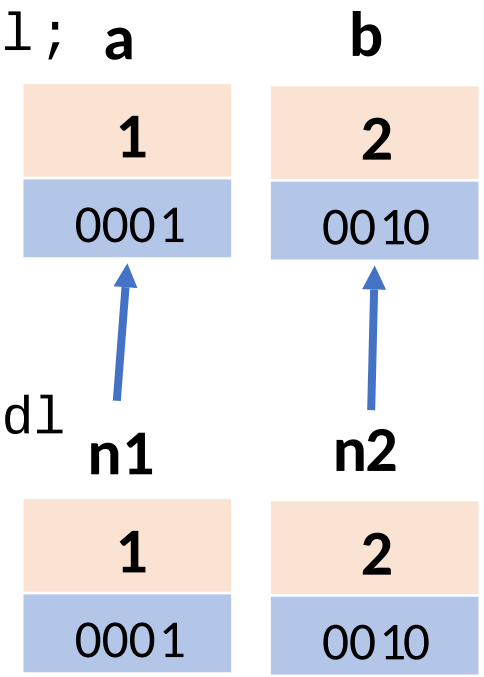


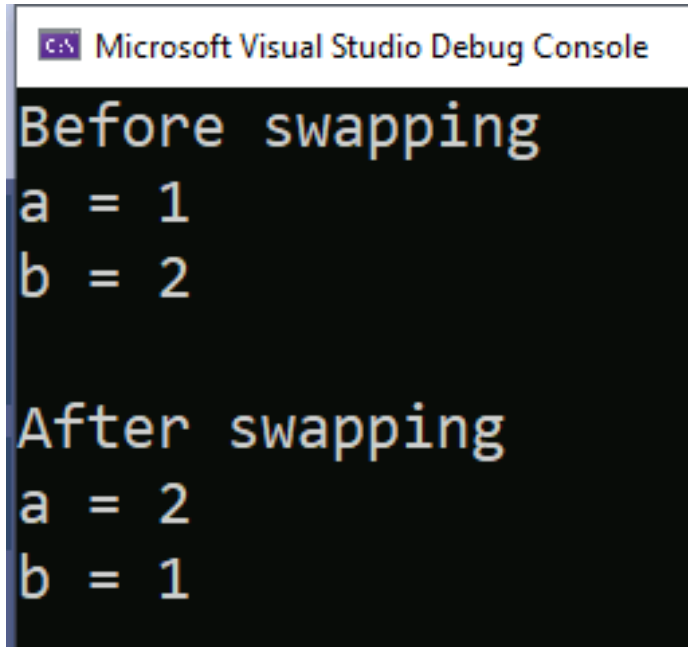
Swapping code: pass by reference..

```
#include <iostream>
using namespace std;
void swap(int*, int*);
int main()
{
    int a = 1, b = 2;
    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    swap(&a, &b);

    cout << "\nAfter swapping" << endl
;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}
```

```
void swap(int* n1, int* n2) {
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

After func call





Summary

- No memory will be allocated for reference
- Pointers, on the other hand, have their own existence in the form of memory. That is, they will be given memory by the compiler.

Programming fundamentals

Lecture 21: *Practice of Pointers*

Recap

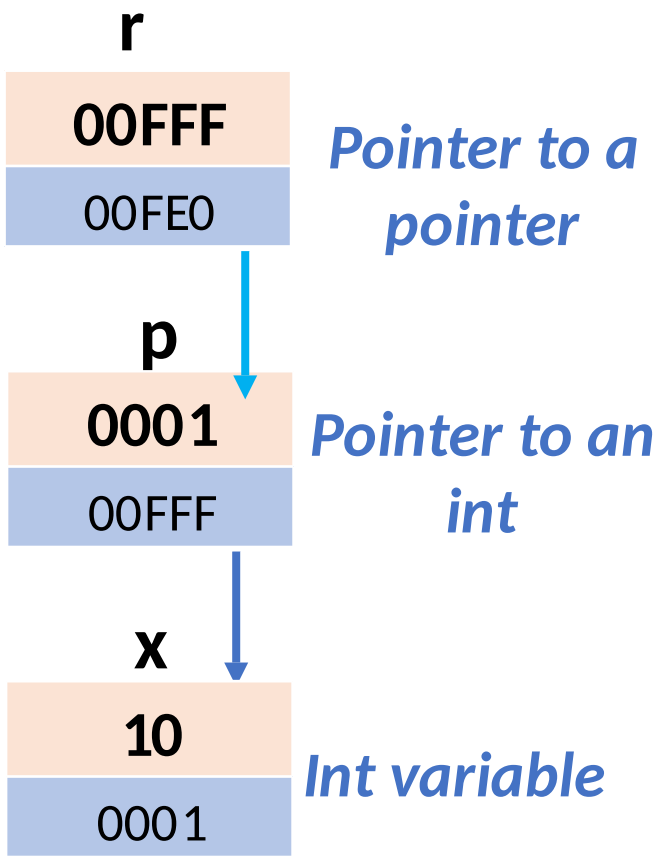
- Array of pointers
- Multiple indirections
- Initializing a pointer
- The Null Pointer Convention
- Dynamic Allocation Using new and delete
- Pass by pointer
- Using pointers in functions

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`++ --`	Suffix/postfix increment and decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
3	`++ --`	Prefix increment and decrement	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(type)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of	
	`new, new[]`	Dynamic memory allocation	
	`delete, delete[]`	Dynamic memory deallocation	
4	`.* ->*`	Pointer to member	Left-to-right
5	`* / %`	Multiplication, division, and remainder	
6	`+ -`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`< <=`	For relational operators < and ≤ respectively	
	`> >=`	For relational operators > and ≥ respectively	
9	`== !=`	For relational = and ≠ respectively	
10	`&`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	
15	`?:`	Ternary conditional	Right-to-left
	`=`	Direct assignment (provided by default for C++ classes)	
	`+= -=`	Assignment by sum and difference	
	`*= /= %=`	Assignment by product, quotient, and remainder	
	`<<= >>=`	Assignment by bitwise left shift and right shift	
	`&= ^= |=`	Assignment by bitwise AND, XOR, and OR	
16	`throw`	Throw operator (for exceptions)	
17	`,`	Comma	Left-to-right

Agenda

- Practice examples related to each core topic
- Lab tasks

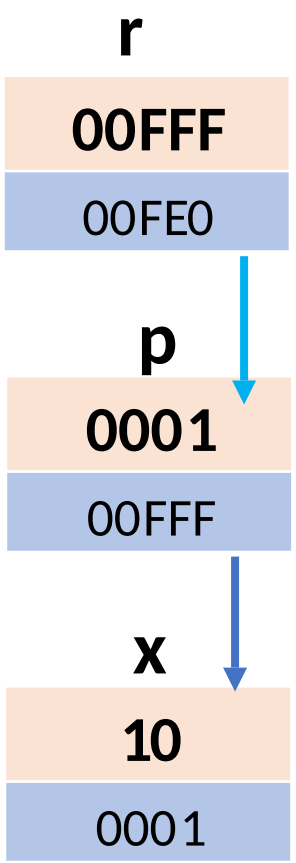
Pointers




```
#include <iostream>
using namespace std;
int main()
{
   int x, y, * p,*q, ** r;
   x = 10; y = 12;
   p = &x;
   q = &y;
   r = &p;
   return 0;
}
```

Contents vs addresses (way to access)

Contents
Address



	Way to access content	Way to access address	*Way to access int value (contents of variable) that a pointer points to*	*Way to access contents of location that a pointer points to*
Pointer to a pointer	r	&r	**r	*r
Pointer to an int	p	&p	*p	-
Int variable	x	&x	-	-

Contents		
Address		
r		
00FFF		
00FE0		
p	q	
0001	0030	
00FFF	00EE0	
x	y	
10	12	
0001	0030	

Value = address

Or

Address = value

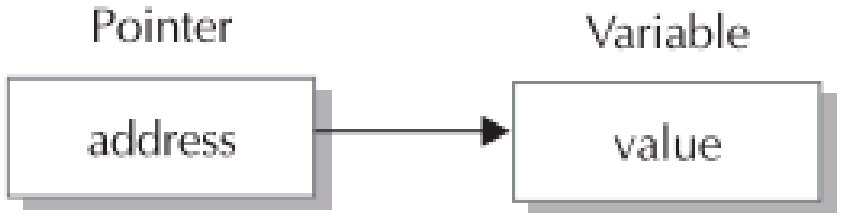
Compile time error!

Not allowed

```
//LHS : value, RHS : address
x = &y;
x = p;
x = q;
*p = &y;
*p = &x;
```

```
//LHS : address, RHS : value
p = x;
q = x;
p = &x;
```

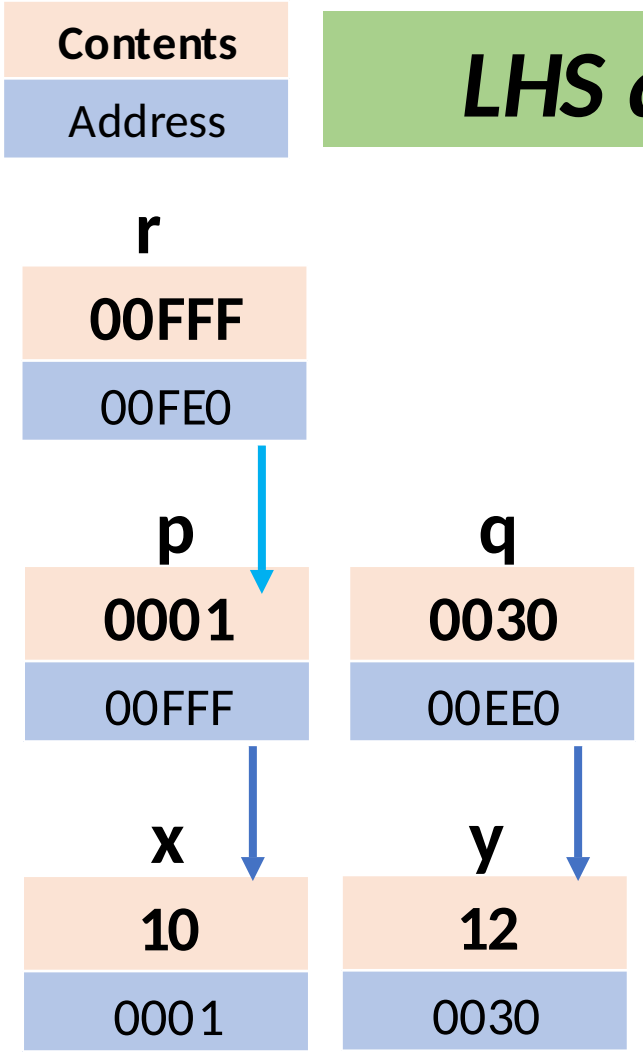
Pointer to a pointer



Single Indirection

Multiple Indirection

LHS and RHS must match in '=' operation!




```
#include <iostream>
using namespace std;
int main()
{
   int x, y, * p,*q, ** r;
   x = 10; y = 12;
   p = &x;
   q = &y;
   r = &p;
   return 0;
}
```

Contents	
Address	
r	
00FFF	
00FE0	
p	q
0001	0030
00FFF	00EE0
x	y
10	12
0001	0030

Value = address

Or

Address = value

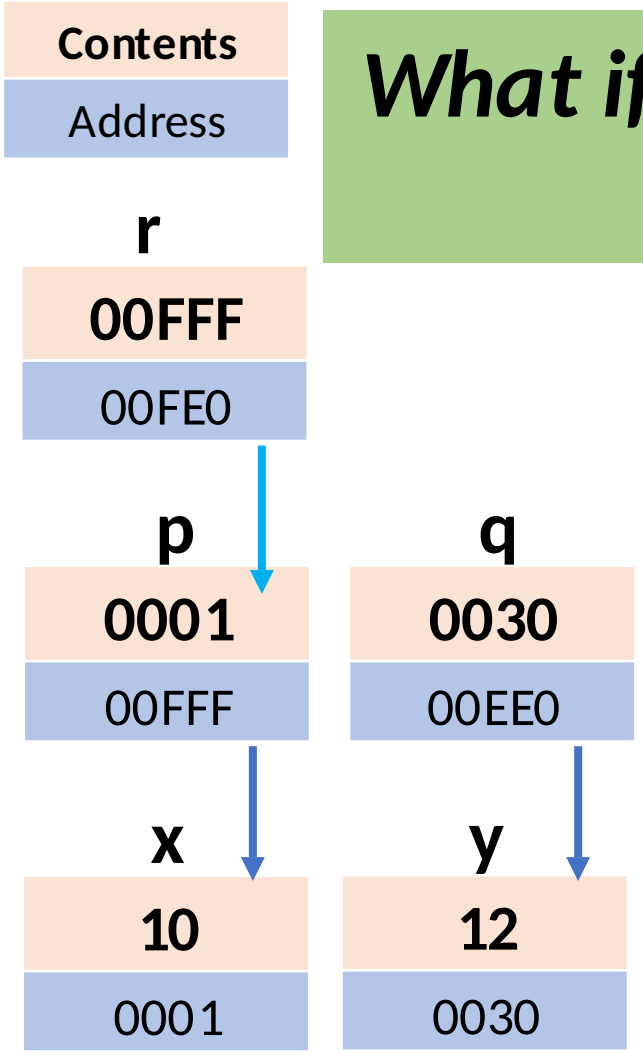
Compile time error!

Not allowed

```
//LHS : value, RHS : address
x = &y;
x = p;
x = q;
*p = &y;
*p = &x;
```

```
//LHS : address, RHS : value
p = x;
q = x;
p = &x;
```

What if RHS and LHS consists of varying TYPES of addresses




```
//LHS : address (pointer to int location),
RHS : address (int location)

p = &x;
r = &x;

//LHS : address (pointer to int location),
RHS : address (address of
pointer(single/multidirectional) to int
location)

p = &q;
p = &r;
```

Compile time error!
Not allowed

```
//type of addresses at RHS and LHS match!
p = &x;
q = p;
r = &p;
```

Correct!

Using different operators with pointers
(Postfix and prefix)

Lab task 1

```
#include <iostream>
using namespace std;
int main()
{
 int var1 = 10, var2 = 5;
 int* p1, * q1, * r1;
 int** g;
 int var3;
 var3 = ++var2;
 p1 = &var1;
 q1 = &var2;
 r1 = &var3;
 g = &r1;
 *p1 = var3++;
 *q1 = ++var1;
//add appropriate lines of code to print
the things mentioned in the table below
 return 0;
}
```

p1	q1	r1	var1	var2	var3
150	250	350	001	002	003

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int var1 = 10, var2 = 5;
6.     int* p1, * q1, * r1;
7.     int** g;
8.     int var3;
9.     var3 = ++var2;
10.    p1 = &var1;
11.    q1 = &var2;
12.    r1 = &var3;
13.    g = &r1;
14.    *p1 = var3++;
15.    *q1 = ++var1;
16.    return 0;
17. }
```

Line#	p1	q1	r1	g	var 1	var2	var 3	
5,6,7,8	G	G	G	G	10	5	G	int * p1, * q1, * r1;
9	G	G	G	G	10	6	6	var3 = ++var2;
10	001	G	G	G	10	6	6	p1=&var1;
11	001	002	G	G	10	6	6	q1=&var2;
12	001	002	003	G	10	6	6	r1=&var3;
13	001	002	003	350	10	6	6	g=&r1
14	001	002	003	350	6	6	7	*p1 = var3++;
15	001	002	003	350	7	7	7	*q1 = ++var1;

Line#	p1	q1	r1	g	var1	var2	var3	
13	001	002	003	350	10	6	6	g=&r1

p1	q1	r1
001	002	003
150	250	350

var1	var2	var3
10	6	6
001	002	003

Line#	p1	q1	r1	g	var 1	var 2	var 3	
14	001	002	003	350	6	6	7	*p1 = var3++;

var1	var2	var3
6	6	7
001	002	003

15	001	002	003	350	7	7	7	*q1 = ++var1;

var1	var2	var3
2 ~~10~~ 6	6	2 ~~6~~ 7
001	002	003

6 1

var1	var2	var3
1 ~~6~~ 7	~~6~~ 7	7
001	002	003

2 7

Postfix and prefix

- The example that we covered in lab luckily deals with ++ operation involving two variables/pointers i.e. expressions like: *a=b++ or *a=++b
- In this case the postfix and prefix would work as they are working
- But, what if the expression with ++ operator only involves one variable/pointer. i.e. expressions like: *a++
- In this case, precedence of operators will be considered by the compiler and postfix would not work as it was meant to work!

Consider the same code with slight adjustment in postfix/prefix expressions

```
#include <iostream>
using namespace std;
int main()
{
    int var1 = 10, var2 = 5;
    int* p1, * q1, * r1;
    int** g;
    int var3;
    var3 = ++var2;

    p1 = &var1;
    q1 = &var2;
    r1 = &var3;

    g = &r1;
    cout << "Before " <<  endl;
    cout << "var1: " << var1 << endl;
    cout << "var2: " << var2 << endl;
    cout << "var3: " << var3 << endl;
    *p1++;
    //*q1 = ++var1;
    cout << "After " << endl;
    cout << "var1: " << var1 << endl;
    cout << "var2: " << var2 << endl;
    cout << "var3: " << var3 << endl;
    return 0;
}
```

Notice a warning that compiler generates!

```
cout << "Before " <<  endl;
cout << "var1: " << var1 << endl;
cout << "var2: " << var2 << endl;
cout << "var3: " << var3 << endl;
*p1++;
//*
cou
cou
cout << "var2: " << var2 << endl;
cout << "var3: " << var3 << endl;
return 0;
```

(local variable) int *p1

Search Online

C6269: Possibly incorrect order of operations: dereference ignored.

There are two operators used in this expression (* and ++)

Precedence of ++ is more than *, thus ++ part will be executed first. Which says: ***p 1++***

- Although you meant to do increment in the value of variable pointed by p1 (var1) through *p1, but, instead of value, the address of p1 is incremented.
- The compiler throws warning to highlight this logical issue.

Remedy …use brackets to overcome that logical mistake!

```
cout << "Before " <<  endl;
cout << "var1: " << var1 << endl;
cout << "var2: " << var2 << endl;
cout << "var3: " << var3 << endl;
(*p1)++;
cout << "After " << endl;
cout << "var1: " << var1 << endl;
cout << "var2: " << var2 << endl;
cout << "var3: " << var3 << endl;
```

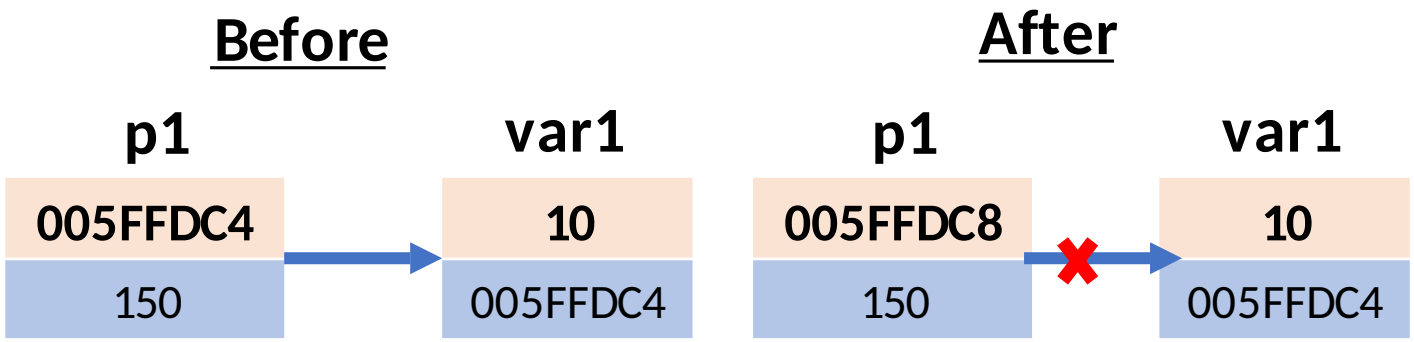
Microsoft Visual Studio Debug Console

```
Before
var1: 10
var2: 6
var3: 6
After
var1: 11
var2: 6
var3: 6
```

In this case, *p1 will be evaluated first because brackets are always evaluated first in any scenario!

Using postfix/prefix expression in cout

In this case, *p1 will be evaluated first and its value will be displayed to the console!




```
cout << "Before " << endl;
cout << "var1: " << var1 << endl;
cout << "var2: " << var2 << endl;
cout << "var3: " << var3 << endl;

cout << *p1++ << endl;

cout << "After " << endl;
cout << "var1: " << var1 << endl;
cout << "var2: " << var2 << endl;
cout << "var3: " << var3 << endl;
```

Microsoft Visual Studio Debug Console

```
Before
*p and p: 10 005FFDC4
var1: 10
var2: 6
var3: 6
10
After
*p and p: -858993460 005FFDC8
var1: 10
var2: 6
var3: 6
```

Array of primitive data type (int, char, double) and **using pointers to access it**

Int array and pointer

```
#include <iostream>
using namespace std;
int main()
{
    int arr[4] = { 1,2,3,4 };
    int* p;
    p = arr;
    cout << "#1" << endl;
    cout << "Address of cell p: " << p << endl;
    cout << "Value at that cell *p: " << *p << endl;
    cout << "Address of pointer itself &p: " << &p << en

    p++;

    cout << "#2" << endl;
    cout << "Address of cell p: " << p << endl;
    cout << "Value at that cell *p: " << *p << endl;
    cout << "Address of pointer itself &p: " << &p << end
    return 0;
}
```

	1	2	3	4
arr	0	1	2	3
	010FFC34	010FFC38	010FFC42	010FFC46

010FFC34	p
010FFC28	

Microsoft Visual Studio Debug Console

```
#1
Address of cell p: 010FFC34
Value at that cell *p: 1
Address of pointer itself &p: 010FFC28
#2
Address of cell p: 010FFC38
Value at that cell *p: 2
Address of pointer itself &p: 010FFC28
```

Using pointer method to access array ***without using square brackets***

```
#include <iostream>
using namespace std;
int main()
{
    int arr[4] = { 1,2,3,4 };
    int* p;
    p = arr;
    cout << "access values using [] notation" << endl;
    for (int i = 0; i < 4; i++)
        cout << arr[i] << " ";

    cout << endl;
    cout << "access values using pointer" << endl;
    for (int i = 0; i < 4; i++)
    {
        cout << *p<< " ";
        p++;
    }
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
access values using [] notation
1 2 3 4
access values using pointer
1 2 3 4
```

Directly accessing any cell of an array with pointer notation

```
#include <iostream>
using namespace std;
int main()
{
   int arr[4] = { 11,2,32,43 };
   int* p;
   p = arr;
   cout << "access value at 4th cell using pointer" << endl;
   p=p+3;
   cout << *p<< " ";
   return 0;
}
```

Access 4th cell by adding 3 into its current address. Thus, 3 is added to base address and pointer is made to now point cell#3 instead of cell#0

Microsoft Visual Studio Debug Console

```
access value at 4th cell using pointer
43
```

Beware of operators' precedence while writing any expression related to pointers!

```
#include <iostream>
using namespace std;
int main()
{
   int arr[4] = { 11,2,32,43 };
   int* p;
   p = arr;
   cout << "access value at 4th cell using pointer" <<
   endl;
   cout << *p+3<< " ";
   return 0;
}
```

Microsoft Visual Studio Debug Console

```
access value at 4th cell using pointer
14
```

*Although you meant to access 4th cell by adding 3 into its current value. But, *p will be evaluated first with cout and 3 is added in the current value (i.e. 11) and we get 14 displayed on the screen*

String, char array and pointers

Using char array to deal *with string* …and using pointers

```
#include <iostream>
using namespace std;
int main()
{

    char c[15];
    char* p;
    cin.getline(c, 15);
    p = c;
    cout << "#1" << endl;
    cout << "Value at that cell *p: " << *p <<
    endl;
    p++;
    cout << "#2" << endl;
    cout << "Value at that cell *p: " << *p <<
    endl;
    return 0;
}
```

c

h	e	l	l	o		S	E	-	F	1	9	\0	G	G
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
010FFC34	010FFC38	…	…	…	…	…	…	…	…	…	…	…	…	…

010FFC34	p
00CFFCFC	

010FFC38	p
00CFFCFC	

Microsoft Visual Studio Debug Console

```
hello SE-F19
#1
Value at that cell *p: h
#2
Value at that cell *p: e
```

- For now focus on char array while practicing with string nature data. However, if you want to use built in *string class* then the procedure is same as described before with only one change:

  For string, you need to explicitly set the char pointer to cell 0
  Base address notation will not work

Using string with pointers

```
#include <string>
#include <iostream>
using namespace std;
int main()
{

    string c;
    getline(cin, c);
    char* p;
    p = &c[0];
    cout << "#1" << endl;
    cout << "Value at that cell *p: " << *p << endl;
    p++;
    cout << "#2" << endl;
    cout << "Value at that cell *p: " << *p << endl;

    return 0;
}
```

c

h	e	l	l	o		S	E	-	F	1	9	\0	G	G
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
010FFC34	010FFC38	…	…	…	…	…	…	…	…	…	…	…	…	…

010FFC34	p
00CFFCFC	

010FFC38	p
00CFFCFC	

Microsoft Visual Studio Debug Console

```
hello SE-F19
#1
Value at that cell *p: h
#2
Value at that cell *p: e
```

Lab task 3 (Bonus)

Write a function ReverseOrder that declares a string and take an input in it. It then r everses its order and displays it. Use only pointer notation (you are not allowed to use array subscript [] notation in the function. You can use more than one pointers and strings in this program if you need any.

Void ReverseOrder();

Example:

i) Input: This is reverse order program

Output: program order reverse is This

ii) Input: This is exercise eight

Output: eight exercise is This

*p

	I		h	a	t	e		m	a	t	h	s	\0		
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	001	002	…	…	…	…	…	…	…	…	…	…	…	…	…

- Idea is to traverse the string from the end, extract words, and side by side print them too
- Extract words from the string and then display them by inverting their order. That is, first display the last word (maths), then second last (hate)..and so on till you reach first word (I)
- Words can be extracted by iterating through an array and checking whether you encounter a space or not. If you encounter a space then that means a word has just been ended
- Iterate through the string using a pointer. Initialize it to the stating cell and then gradually move forward or backward (according to your designed algorithm)

Steps Logic..

*p

	I		h	a	t	e		m	a	t	h	s	\0		
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	001	002	…	…	…	…	…	…	…	…	…	…	…	…	…

- Iterate through an array and set the pointer to the end of an array

  Or directly set the pointer to the end of an array

  Or start accessing or iterating through an array from the end

*p

	I		h	a	t	e		m	a	t	h	s	\0		
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	001	002	…	…	…	…	…	…	…	…	…	…	…	…	…

- Continue decrementing pointer till you reach at the start of the last word
- Notice that the start of this last word will be indicated by a space ‘ ’ character
- Hint: you can use if statement to check if you encounter a space while traversing

- Once you reach that point then it is now a time to display that last word
- Iterate through the string using pointer till you read and display that word
- Don’t forget to store index location where you encounter a space..just to continue with your algorithm after printing last word

*p

	I		h	a	t	e		m	a	t	h	s	\0		
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	001	002	…	…	…	…	…	…	…	…	…	…	…	…	…

- Continue with the same algorithm/logic and reach a point at where you encounter a second space (while moving backward)
- Once you reach that point then it is now a time to display that second last word
- Iterate through the string using pointer till you read and display that word
- Don't forget to store index location where you encounter a space..just to continue with your algorithm after printing second last word

- You can continue moving backward till you index is greater than equal to 0!
- In short, this will help in checking if you have reached the first character of your input string!

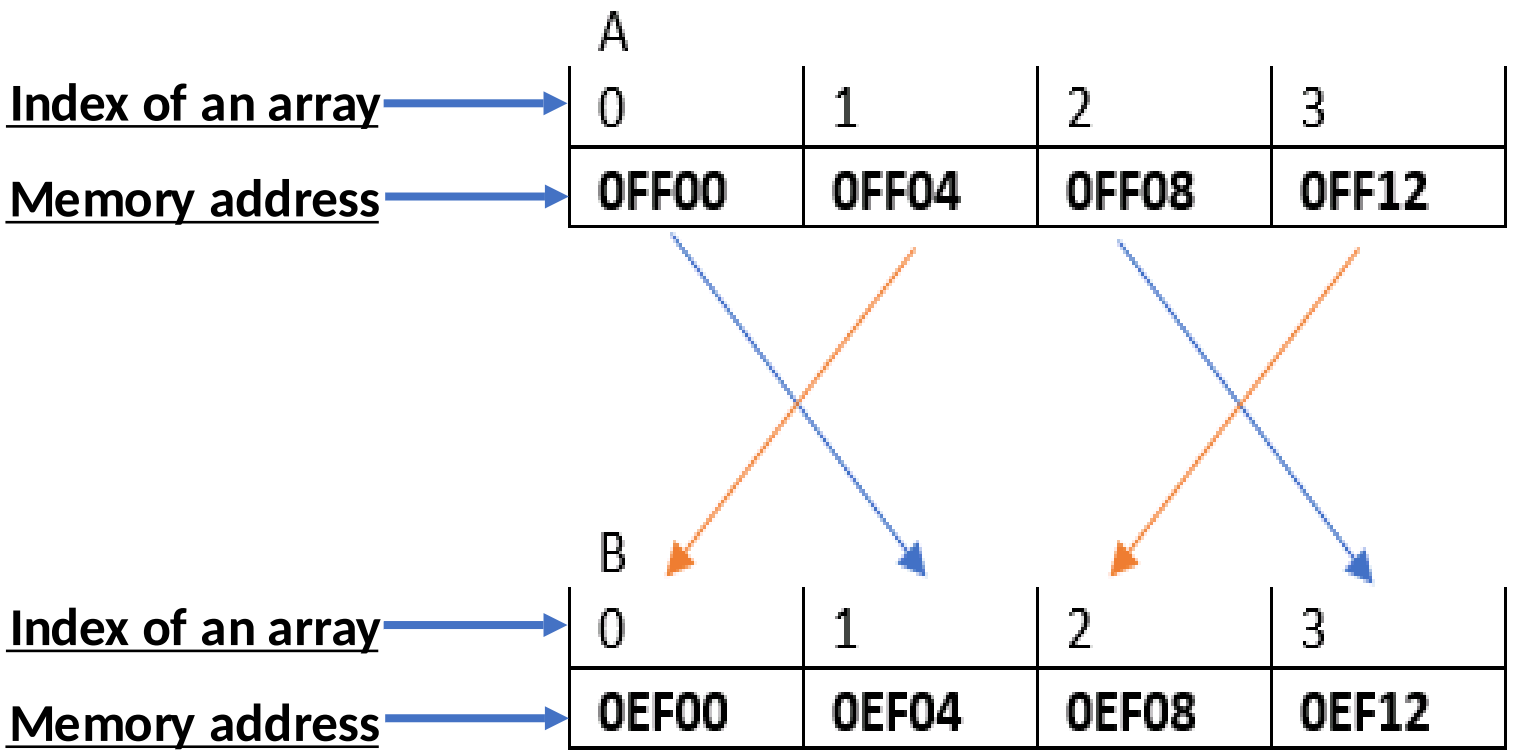
Code (try yourself first, use this solution only for reference)

```
#include <iostream>
#include <cstring>
using namespace std;
void reverse(char* sentence)
{
    int index = strlen(sentence) - 1, hold, last = '\0';
    /*For the 1st iteration last is `\0` for all others it is ` `*/
    while (index >= 0)
    {
      while (index >= 0 && sentence[index] != ' ')
            index--;
      hold = index - 1;  // This keeps track of the last character of
preceding word
        index++; //character after space
        while (sentence[index] != last)
        {
            cout << sentence[index];
            index++;
        }
        last = ' ';
        index = hold;
        /* Dont print space after 1st word*/
        if (index > 0)
            cout << " ";
    }
}
```

```
int main()
{
    char* sentence = new char[256];
    cin.getline(sentence, 256);
    reverse(sentence);
    delete[] sentence;
}
```

Array of pointers

Lab task 2



Task was to create a link between two arrays.

- *To create a link, it is obvious that A must be a pointer only then we can point its cells to somewhere.*
- *However, there is no restriction imposed on B. B can be an integer array of an array of pointers to integer*

Case 1: int ** A[4]; int * B[4]

A

0	1	2	3
0E000	0E004	0E008	0E012

→ int **A[4];

Since A is an array of pointers to a pointer, thus its **contents** would be a pointer i.e. ***an address to pointer***

B

0	1	2	3
0FF00	0FF04	0FF08	0FF12

→ int *B[4]

Since B is an array of pointers, thus its **contents** would be **an address to an integer value**

Both the arrays A and B will have a dedicated memory address (e.g. 0E000, 0FF00 etc.), allocated by the compiler.

A	.. A8	..A4	.. B0	.. AC
	0	1	2	3
	0115F9B	.. C0	.. C4	.. C8
B	0	1	2	3
	0115F9A4	.. A8	.. AC	.. B0

Microsoft Visual Studio Debug Console

```
Addresses
A[0]: 0115F9BC
B[0]: 0115F9A4
A[1]: 0115F9C0
B[1]: 0115F9A8
A[2]: 0115F9C4
B[2]: 0115F9AC
A[3]: 0115F9C8
B[3]: 0115F9B0
```

```
Contents after setting the contents of A
A[0]: 0115F9A8
A[1]: 0115F9A4
A[2]: 0115F9B0
A[3]: 0115F9AC
```

What if?

```
int* A[4];
int* B[4];

instead of
int** A[4];
int* B[4];
```

Compile time error!

The addresses' types also match in LHS and RHS!

A

0	1	2	3
0E000	0E004	0E008	0E012

→ int ** A[4];

Since A is an array of pointers, thus its **contents** would be a pointer i.e. ***an address to pointer to an int location***

B

0	1	2	3
0FF00	0FF04	0FF08	0FF12

→ Int * B[4]

Since B is an array of integers, thus its **contents** would be **pointer i.e. an address to a int location**

```
A[0] = &B[1];
```

Address to a pointer **Address to an integer**

Case 2: int * A[4]; int B[4]

A

0	1	2	3
0E000	0E004	0E008	0E012

→ int * A[4];

Since A is an array of pointers, thus its **contents** would be a pointer i.e. ***an address to int location***

B

0	1	2	3
0FF00	0FF04	0FF08	0FF12

→ int B[4]

Since B is an array of integers, thus its **contents** would be **an integer value**

Both the arrays A and B will have a dedicated memory address (e.g. 0E000, 0FF00 etc.), allocated by the compiler.

Practice questions v10
Q1

Dynamic arrays

new and delete keyword

- C++ contains two operators, new and delete, that perform the functions of allocating and freeing memory.
- Their general forms are shown here:

```
pointer-var = new var-type;
delete pointer-var;
```

- Here, pointer-var is a pointer of type var-type. The new operator allocates sufficient memory to hold a value of type var-type and returns a pointer to it. Any valid data type can be allocated using new.
- The delete operator frees the memory pointed to by pointer-var. Once freed, this memory can be reallocated to different purposes by a subsequent new allocation request.

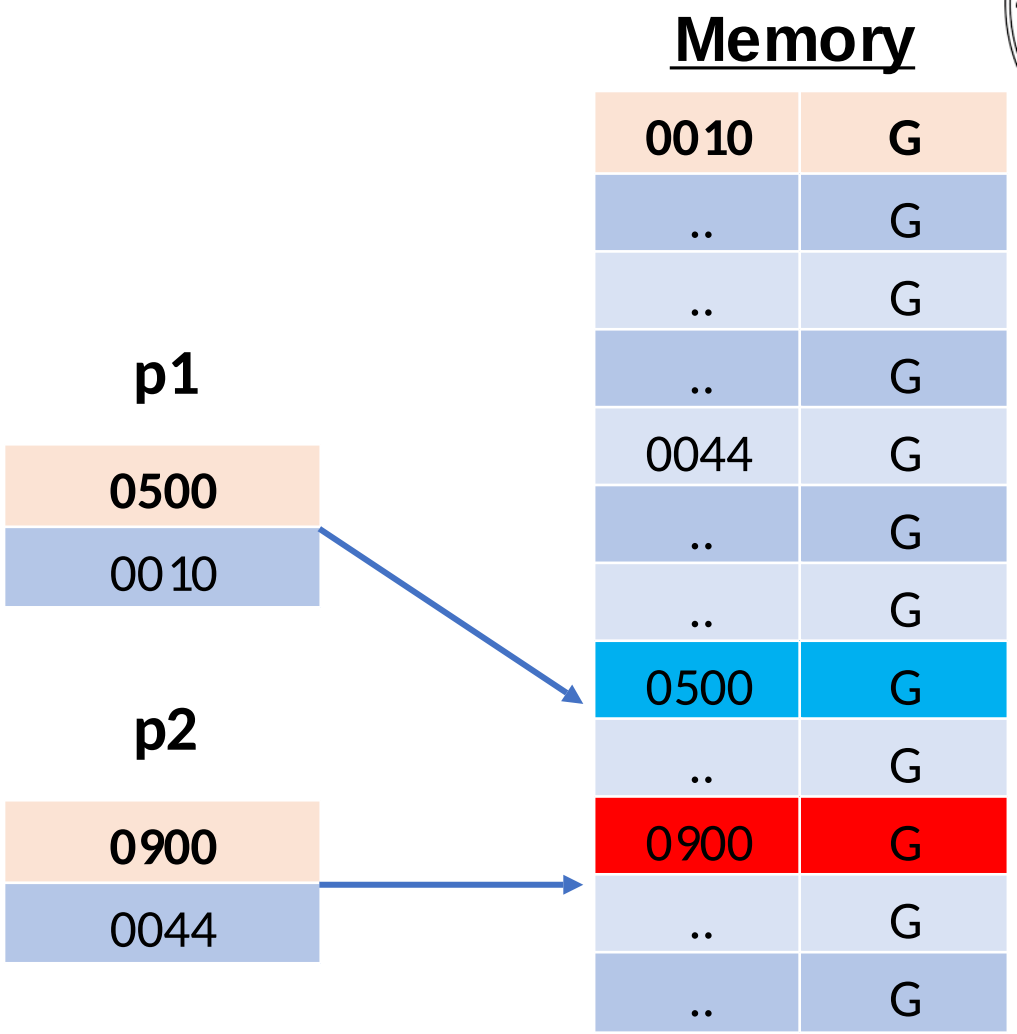
Dynamic memory allocation

```
#include <iostream>
using namespace std;
int main()
{
   int* p;
   p = new int; // allocate memory for int
   *p = 20; // assign that memory the value 20
   cout << *p; // prove that it works by displaying value
   delete p; // free the memory
   return 0;
}
```

Practice questions v10 Q2

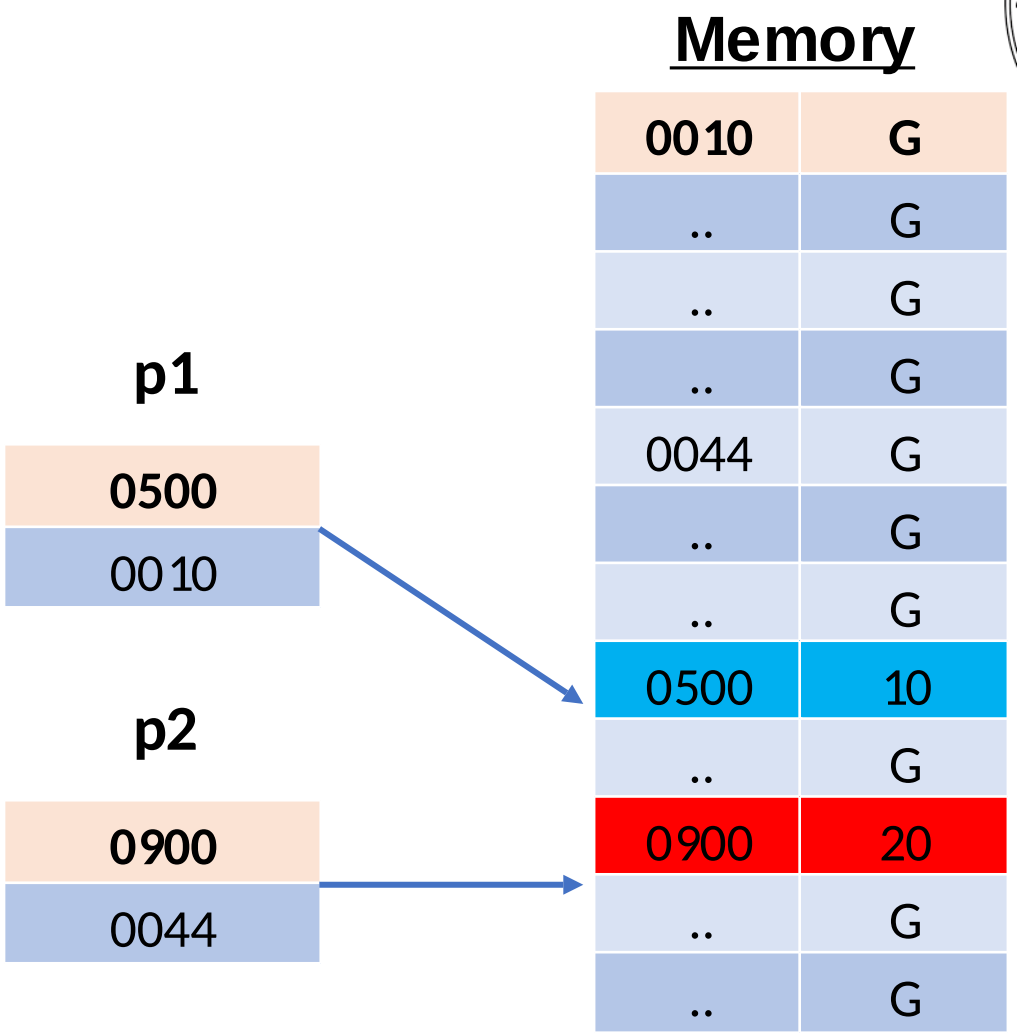
Q2

```
#include <iostream>
using namespace std;
int main()
{
    int* p1, * p2;
    p1 = new int;
    p2 = new int;
    *p1 = 10;
    *p2 = 20;
    cout << *p1 << " " << *p2 << endl;
    p1 = p2;
    cout << *p1 << " " << *p2 << endl;
    *p1 = 30;
    cout << *p1 << " " << *p2 << endl;

}
```



Q2

```
#include <iostream>
using namespace std;
int main()
{
    int* p1, * p2;
    p1 = new int;
    p2 = new int;
    *p1 = 10;
    *p2 = 20;
    cout << *p1 << " " << *p2 << endl;
    p1 = p2;
    cout << *p1 << " " << *p2 << endl;
    *p1 = 30;
    cout << *p1 << " " << *p2 << endl;

}
```

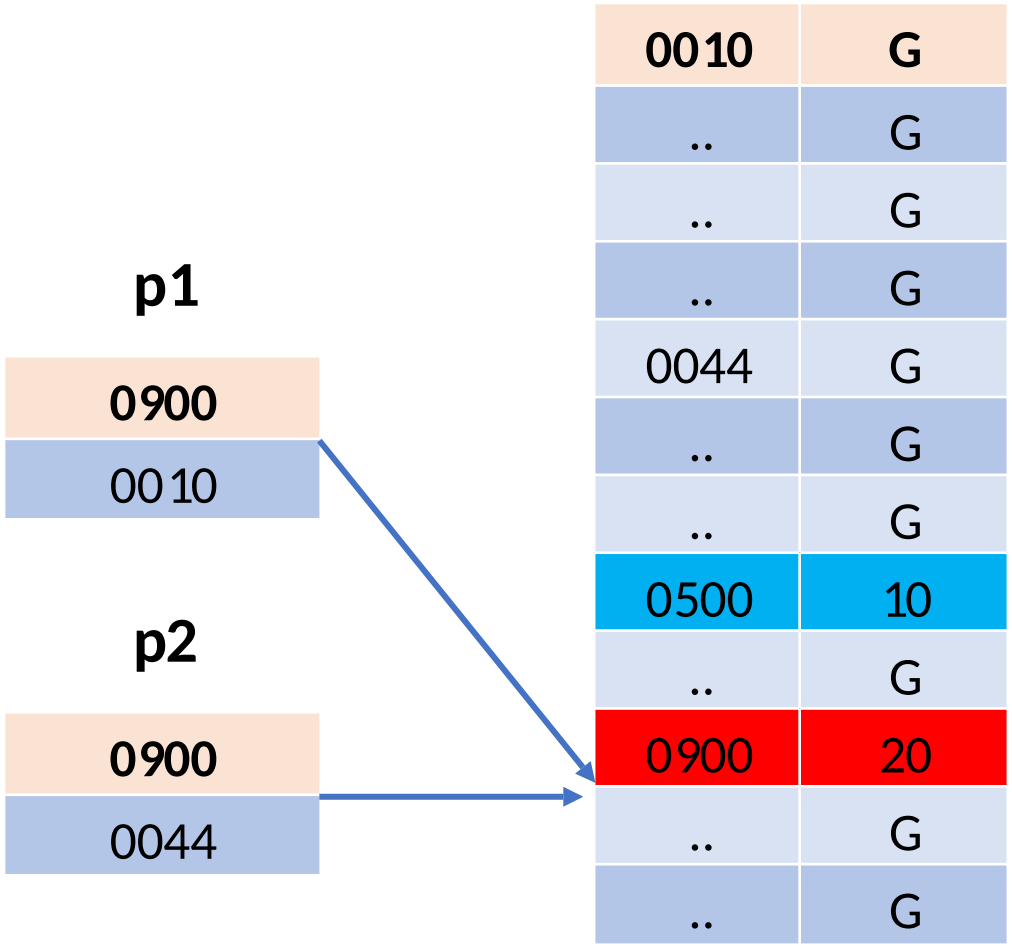


Q2

```
#include <iostream>
using namespace std;
int main()
{
    int* p1, * p2;
    p1 = new int;
    p2 = new int;
    *p1 = 10;
    *p2 = 20;
    cout << *p1 << " " << *p2 << endl;
    p1 = p2;
    cout << *p1 << " " << *p2 << endl;
    *p1 = 30;
    cout << *p1 << " " << *p2 << endl;

}
```

Output:

1. 20



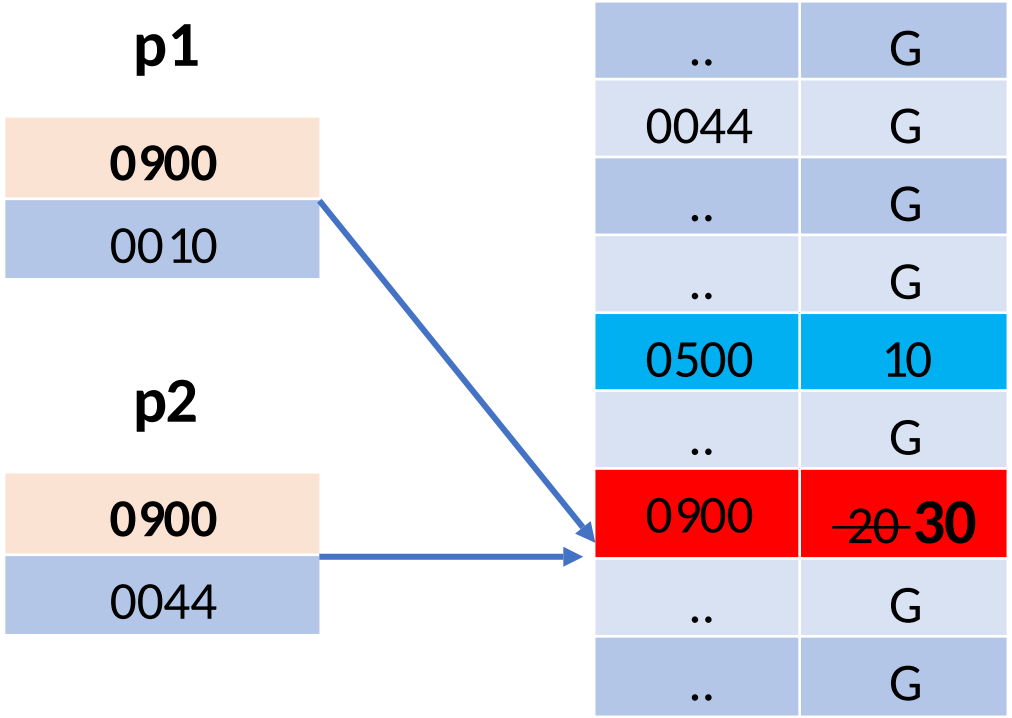
Q2

```
#include <iostream>
using namespace std;
int main()
{
    int* p1, * p2;
    p1 = new int;
    p2 = new int;
    *p1 = 10;
    *p2 = 20;
    cout << *p1 << " " << *p2 << endl;
    p1 = p2;
    cout << *p1 << " " << *p2 << endl;
    *p1 = 30;
    cout << *p1 << " " << *p2 << endl;

}
```

Output:

1. 20

20 20

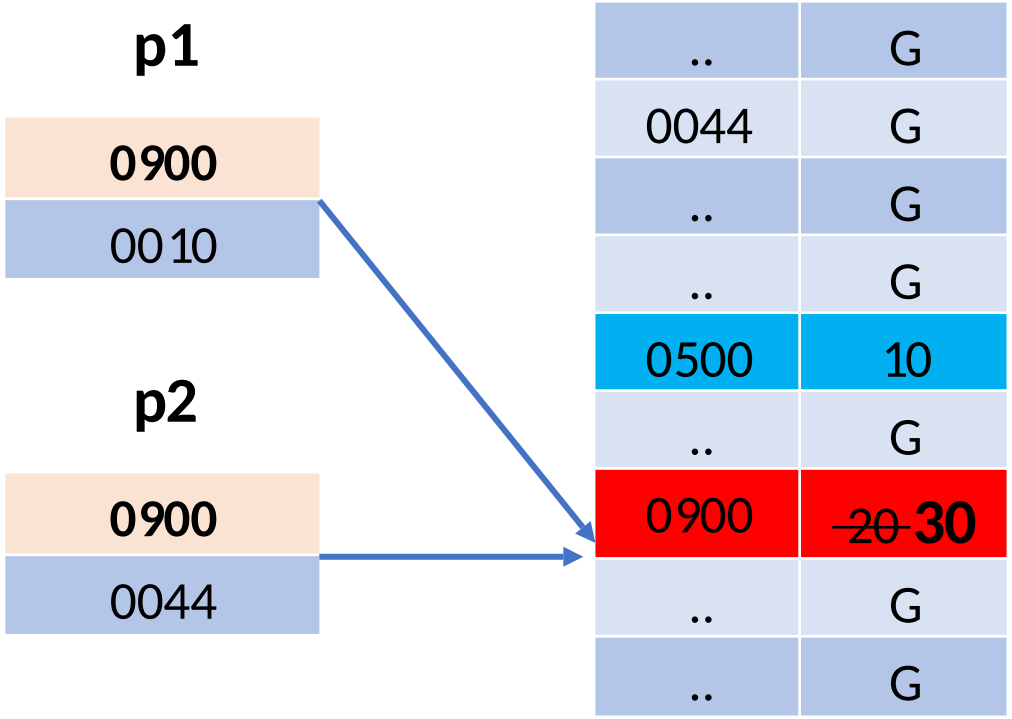


Q2

```
#include <iostream>
using namespace std;
int main()
{
    int* p1, * p2;
    p1 = new int;
    p2 = new int;
    *p1 = 10;
    *p2 = 20;
    cout << *p1 << " " << *p2 << endl;
    p1 = p2;
    cout << *p1 << " " << *p2 << endl;
    *p1 = 30;
    cout << *p1 << " " << *p2 << endl;

}
```

Output:

1. 20
2. 20
30 30



Allocating arrays

- Everything is same as we have done for allocating one memory location dynamically.
- Now we are going to explore how to allocate arrays
- Syntax for allocating and deallocating memory for arrays is:

```
pointer-var = new type [size];

delete [ ] pointer-var;
```

```
#include <iostream>
using namespace std;
int main()
{
   int* p;
   int i;
   p = new int[10]; // get a 10-element array
   for (i = 0; i < 10; i++)
     p[i] = i;
   delete[] p; // delete the entire array
   return 0;
}
```

	0	1	2	3	4	5	6	7	8	9
p	0000	0004	0008	0012	0016	0020	0024	0028	0032	0036

You can now have user defined size array with dynamic allocation

```
#include <iostream>
using namespace std;
int main()
{
   int* p;
   int size = 0;
   cout << "Enter the size:" << endl;
   cin >> size;
   p = new int[size]; // get a 10-element array
   for (int i = 0; i < size; i++)
   p[i] = i;
   delete[] p; // delete the entire array
   return 0;
}
```

Microsoft Visual Studio Debug Console

```
Enter the size:
5
```

Make sure to access only those memory that you allocated, otherwise your compiler will give exception in this case

Practice questions v10 Q4

Q4

```
#include <iostream>
using namespace std;
int main()
{
    int array_size = 10;
    int* a;
    a = new int[array_size];
    int i;
    for (i = 0; i < array_size; i++)
          *(a + i) = i;
    for (i = 0; i < array_size; i++)
          cout << a[i] << " ";
    cout << endl;
}
```

G	G	G	G	G	G	G	G	G	G
0	1	2	3	4	5	6	7	8	9
001	002	003	004	005	006	007	008	009	010

001
0FF0

a

Q4

```
#include <iostream>
using namespace std;
int main()
{
    int array_size = 10;
    int* a;
    a = new int[array_size];
    int i;
    for (i = 0; i < array_size; i++)
          *(a + i) = i;
    for (i = 0; i < array_size; i++)
          cout << a[i] << " ";
    cout << endl;
}
```

G 0	G	G	G	G	G	G	G	G	G
0	1	2	3	4	5	6	7	8	9
001	002	003	004	005	006	007	008	009	010

001
0FF0

Iteration#0: i=0

a

***(a+i) = i**

***(001+0) = 0**

***(001) = 0**

Value at 001 = 0

Q4

```
#include <iostream>
using namespace std;
int main()
{
    int array_size = 10;
    int* a;
    a = new int[array_size];
    int i;
    for (i = 0; i < array_size; i++)
          *(a + i) = i;
    for (i = 0; i < array_size; i++)
          cout << a[i] << " ";
    cout << endl;
}
```

0	~~G~~ 1	G	G	G	G	G	G	G	G
0	1	2	3	4	5	6	7	8	9
001	002	003	004	005	006	007	008	009	010

a	001
	0FF0

Iteration#1: i=1

```
*(a+i)      =
*(001+1)    =   1
*(002)      =   1
Value at 002 =   1
```

And the procedure continues till i=9…

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
001	002	003	004	005	006	007	008	009	010

a	**001**
	0FF0

Pointers in functions

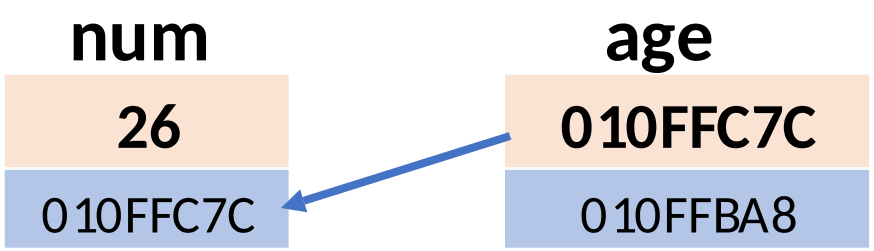
C++ allows you to pass a pointer to a function.

Pass by Pointe

Example 1:

pointer = variable's address *assignment*

```
#include <iostream>
using namespace std;
void printAge(int* age);

int main() {
    int num = 26;
    int* age;
    age = &num;
    return 0;
}
```



Example 1

```
#include <iostream>
using namespace std;
void func(int *age);

int main() {
    int num=26;
    cout << "num : " << num << endl;
    cout << "Address of num : " << &num <<
endl;
    func(&num);
    return 0;
}

void func(int *age) {
    cout << "Age received : " << age << endl;
    cout << "Address of age : " << &age <<
endl;
}
```

num	age
26	010FFC7C
010FFC7C	010FFBA8

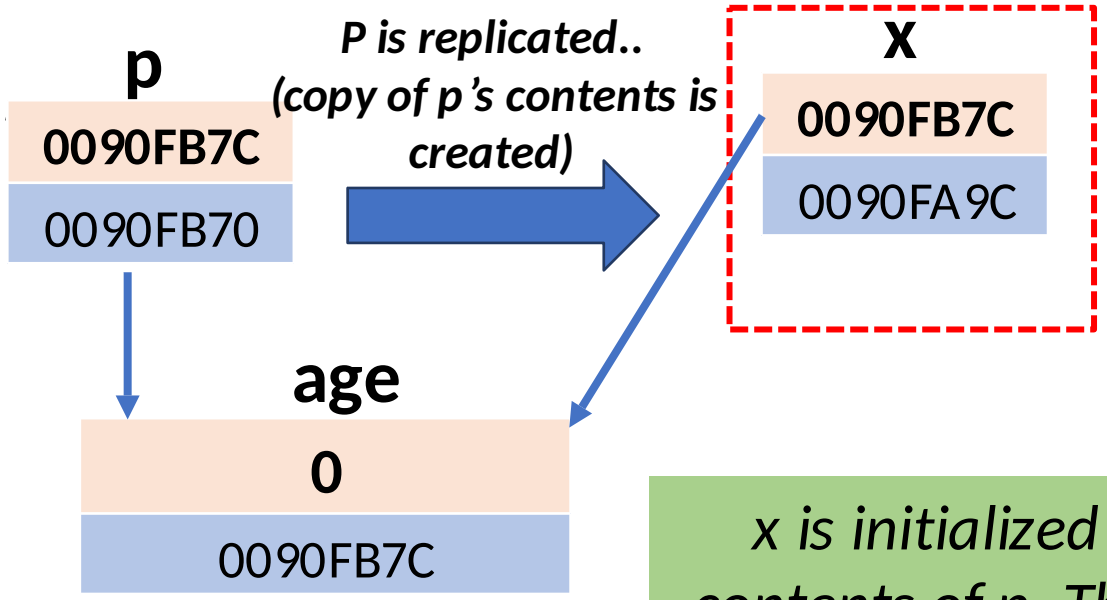
```
age = &num;
pointer = address;
```

Microsoft Visual Studio Debug Console

```
num: 26
Address of num: 010FFC7C
Age received: 010FFC7C
Address of age: 010FFBA8
```

Example 2:

pointer = pointer *assignmen*



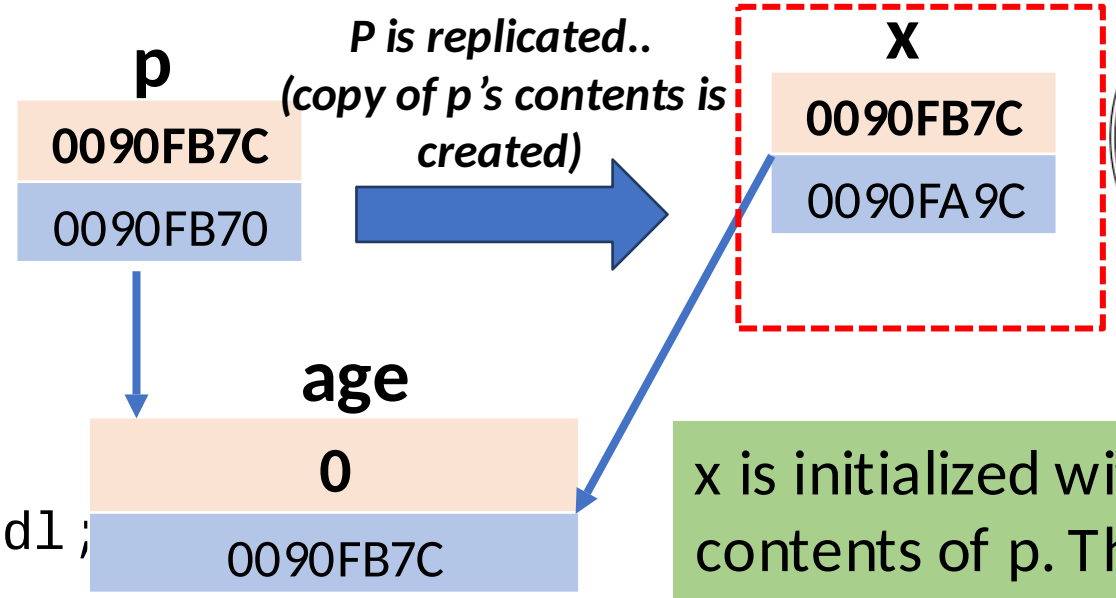

```
#include <iostream>
using namespace std;

int main()
{
    int age = 0; int* p; int* x;
    p = &age;
    x = p;
    return 0;
}
```

x is initialized with contents of p. That is, x also points where p was pointing. Because contents of pointer decide where to point!

```
#include <iostream>
using namespace std;
void printAge(int *x);

int main() {
    int age=0; int* p;
    p = &age;
    cout << "&age : " << &age << endl;
    cout << "p : " << p << endl;
    cout << "&p : " << &p << endl;
    printAge(p);
    return 0;
}

void printAge(int *x) {
    cout << "x : " << x << endl;
    cout << "&x : " << &x << endl;
}
```

x = p;

pointer = pointer;



x is initialized with contents of p. That is, x also points where p was pointing

Microsoft Visual Studio

```
&age: 0090FB7C
p: 0090FB7C
&p: 0090FB70
x: 0090FB7C
&x: 0090FA9C
```

Practice questions v10 Q1

1. What is the output of this program?

```
#include <iostream>
using namespace std;
int main()
{
    int a[4] = { 1, 2, 3, 4 };
    cout << *(a + 2) << (*(a) + (2*5))
    return 0;
}
```

A.3 1 7

B. 31 1

C. 8 8 8

D. Compile time error

	0	1	2	3
a	1	2	3	4
	0000	0001	0002	0003

a : base address : 0000

*(a + 2)

*(0000 + 2)

*(0002)

3

(*(a) + (2*5))

(*(0000) + (10))

(1 + 10)

(1 + 10)

11

```
#include <iostream>
using namespace std;
void mystery1(char*, const char*);
int main()
{
    char string1[80];
    char string2[80];
    cout << "Enter two strings: ";
    cin >> string1 >> string2;
    mystery1(string1, string2);
    cout << string1 << endl;
}
void mystery1(char* s1, const char* s2)
{
    while (*s1 != '\0')
        ++s1;
    for (; (*s1 = *s2); ++s1, ++s2)
    {
    }
}
```

string1

H	e	l	l	o	\0					
0	1	2	3	4	5	6	7	8	-	79
001	002	00	004	005	006	007	008	009	010	011

string2

S	E	\0								
0	1	2	3	4	5	6	7	8	-	79
011	012	013	014	015	016	017	018	019	020	021

Formal param1,param2,..param n func call (Passed param1,param2,..param n)

s1,s2 mystery1 (string1, string 2)

Base address copied to formal parameters

001
00CFFCFC

s1

011
0EEECFC

s2

```
void mystery1(char* s1, const char* s2)
{
    while (*s1 != '\0')
          ++s1;
    for (; (*s1 = *s2); ++s1, ++s2)
    {
    }
}
```

001
00CFFCFC

s1

s1

string1

H	e	l	l	o	\0					
0	1	2	3	4	5	6	7	8	-	79
001	002	003	004	005	006	007	008	009	010	011

011
00EEECFC

s2

s2

string2

S	E	\0								
0	1	2	3	4	5	6	7	8	-	79
011	012	013	014	015	016	017	018	019	020	021

```
void mystery1(char* s1, const char* s2)
{
    while (*s1 != '\0')
          ++s1;
    for (; (*s1 = *s2); ++s1, ++s2)
    {
    }
}
```

006
00CFFCFC

s1

011
00EEECFC

s2

s1

string1

H	e	l	l	o	\0					
0	1	2	3	4	5	6	7	8	-	79
001	002	003	004	005	006	007	008	009	010	011

s2

string2

S	E	\0								
0	1	2	3	4	5	6	7	8	-	79
011	012	013	014	015	016	017	018	019	020	021

```
void mystery1(char* s1, const char* s2)
{
    while (*s1 != '\0')
          ++s1;
    for (; (*s1 = *s2); ++s1, ++s2)
    {
    }
}
```

006
00CFFCFC

s1

011
00EEECFC

s2

s1

string1

H	e	l	l	o	\0					
0	1	2	3	4	5	6	7	8	-	79
001	002	003	004	005	006	007	008	009	010	011

s2

string2

S	E	\0								
0	1	2	3	4	5	6	7	8	-	79
011	012	013	014	015	016	017	018	019	020	021

Iteration#1

```
*s1 = *s2
Value at 006 = value at
Value at 006 = 'S'
```

```
for (; (*s1 = *s2); ++s1, ++s2)
{
}
```

006
00CFFCFC

s1

s1

string1

H	e	l	l	o	~~\0~~ S					
0	1	2	3	4	5	6	7	8	-	79
001	002	003	004	005	006	007	008	009	010	011

011
00EEECFC

s2

s2

string2

S	E	\0			
0	1	2	3	-	79
011	012	013	014	020	021

*s1 = *s2
Is an expression. Expressions in C/C++ evaluates to values, and in this case it returns the value assigned to *s1. When the '\0' is assigned to *s1, the expression evaluates to 0 which is false clearly.

Iteration#2

```
for (; (*s1 = *s2); ++s1, ++s2)
{
}
```

007
00CFFCFC

s1

s1

string1

H	e	l	l	o	S					
0	1	2	3	4	5	6	7	8	-	79
001	002	003	004	005	006	007	008	009	010	011

012
00EEECFC

s2

s2

string2

S	E	\0								
0	1	2	3	4	5	6	7	8	-	79
011	012	013	014	015	016	017	018	019	020	021

Iteration#3

```
for (; (*s1 = *s2); ++s1, ++s2)
{
}
```

*s1 = *s2 true

Value at 007 = value at 012

Value at 007 = 'E' This expression returns 'E'

007	
00CFFCFC	s1

s1

string1

H	e	l	l	o	S	E				
0	1	2	3	4	5	6	7	8	-	79
001	002	003	004	005	006	007	008	009	010	011

012	
00EEECFC	s2

s2

string2

S	E	\0								
0	1	2	3	4	5	6	7	8	-	79
011	012	013	014	015	016	017	018	019	020	021

Iteration#4

```
for (; (*s1 = *s2); ++s1, ++s2)
{
}
```

008
00CFFCFC

s1

s1

string1

H	e	l	l	o	S	E				
0	1	2	3	4	5	6	7	8	-	79
001	002	003	004	005	006	007	008	009	010	011

013
00EEECFC

s2

s2

string2

S	E	\0								
0	1	2	3	4	5	6	7	8	-	79
011	012	013	014	015	016	017	018	019	020	021

Iteration#5

```
*s1     =     *s2
```
false

```
Value at 008  =  value at
Value at 008  =  '\0'
```
This expression returns '\0' i.e. 0

```
for (; (*s1 = *s2); ++s1, ++s2)
{
}
```

Conditional part evaluated to false, thus stop the loop!

008
00CFFCFC

s1

013
00EEECFC

s2

s1

string1

H	e	l	l	o	S	E				
0	1	2	3	4	5	6	7	8	-	79
001	002	003	004	005	006	007	008	009	010	011

s2

string2

S	E	\0								
0	1	2	3	4	5	6	7	8	-	79
011	012	013	014	015	016	017	018	019	020	021

Output

string1

H	e	l	l	o	S	E				
0	1	2	3	4	5	6	7	8	-	79
001	002	003	004	005	006	007	008	009	010	011

```
#include <iostream>
using namespace std;
void mystery1(char*, const char*);
int main()
{
   char string1[80];
   char string2[80];
   cout << "Enter two strings: ";
   cin >> string1 >> string2;
   mystery1(string1, string2);
   cout << string1 << endl;
}
```

Microsoft Visual Studio Debug Console

```
Enter two strings: Hello SE
0
HelloSE
```

Programming fundamentals

Lecture 22: *2D dynamic array using pointers, using functions with dynamic arrays*

Recap

- Dynamic 2D array

Agenda

- 2D dynamic array

2D dynamic array
Doing everything in main()

2D array (static)

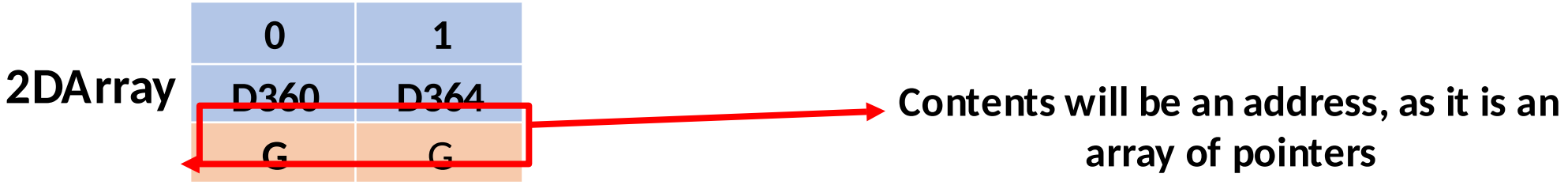
```
#include <iostream>
using namespace std;
void print(int arr[][3])
{
    for (int i = 0; i < 2; i++)
    {
        for (int j=0 ; j<3 ; j++)
        {
            cout << "Value arr[" << i <<
            ","<<j<< "]: " << arr[i][j]
            << " " << endl;
            cout << "Address arr[" << i
            << "," << j << "]: " << &arr
            [i][j] << " " << endl;
        }
    }
}
int main()
{
    int arr[2][3] = {1,2,3,4,5,6};
    print(arr);
    return 0;
}
```

0,0	0,1	0,2
007CFE5C	007CFE60	007CFE64
1	*2*	*3*
1,0	1,1	1,2
007CFE68	007CFE6C	007CFE70
4	*5*	*6*

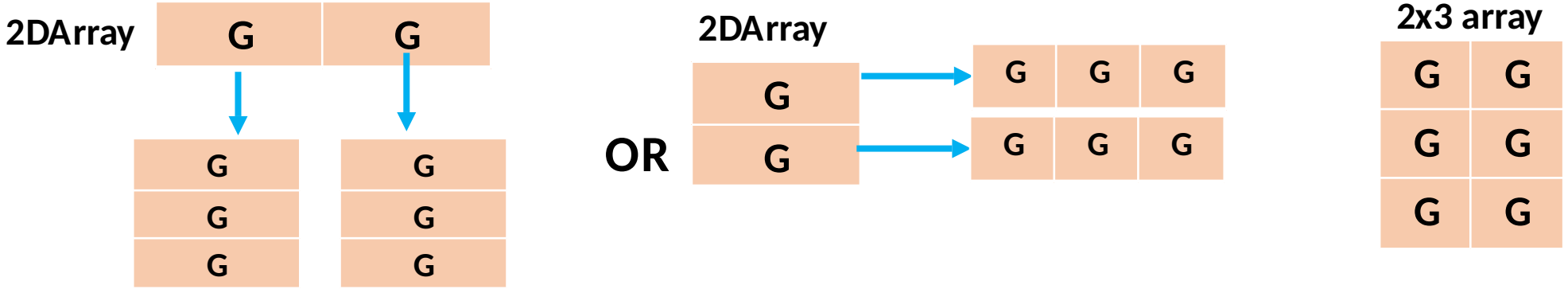
0	007CFE5C	1
1	007CFE60	2
2	007CFE64	3
3	007CFE68	4
4	007CFE6C	5
5	007CFE70	6
6	007CFE74	-858993460
7	007CFE78	1636287196
8	007CFE7C	*8191644*
9	007CFE80	*3485315*
10	007CFE84	1
11	007CFE88	*12214168*
12	007CFE8C	*12232232*
13	007CFE90	1
14	007CFE94	*12214168*
15	……	

2D array

- Can be thought of as **an array of arrays**
- We first create a one-dimensional dynamic array of pointers of type *int**, which is the type for a one-dimensional array of *ints*.



- Then you create a dynamic array of *ints* for each indexed variable of the array of pointers.



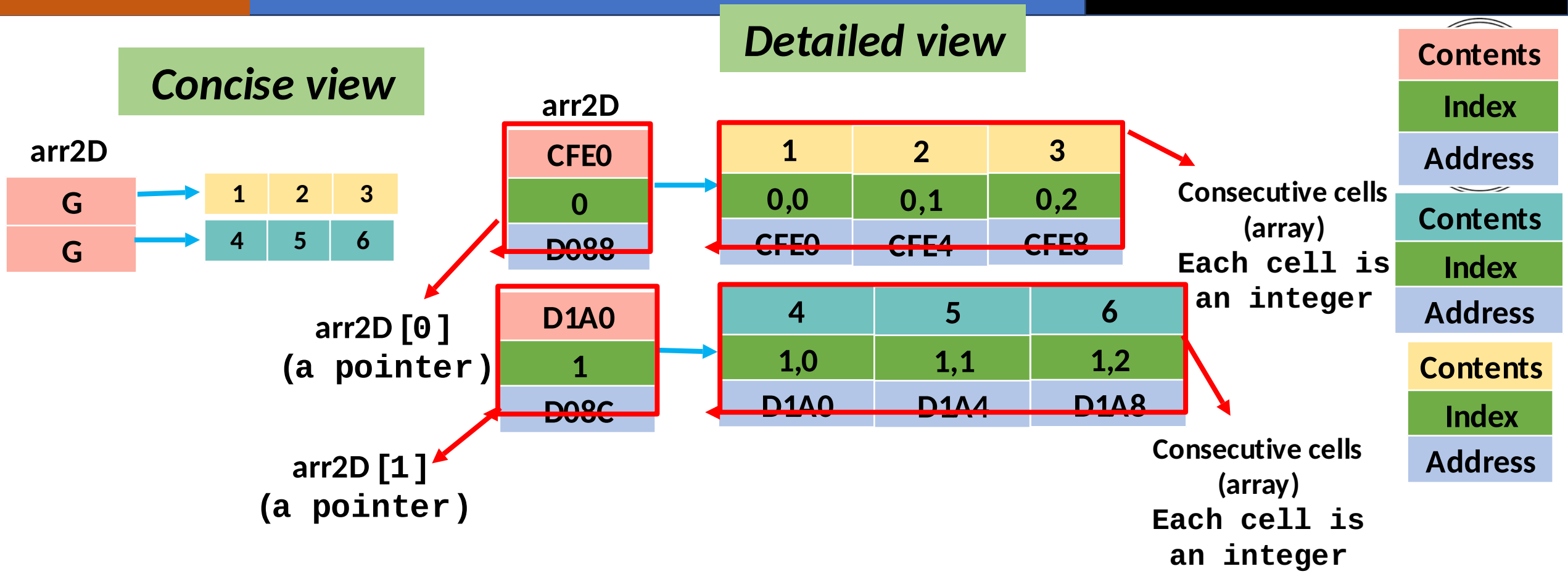
```
#include<iostream>
using namespace std;
int main() {
int size = 3;
int row, col;
cout << "Enter dimension 1 (# of rows:) "
<< endl;
cin >> row;
cout << "Enter dimension 2 (# of cols:) "
<< endl;
cin >> col;

int **arr2D;
arr2D = new int*[row];
cout << "Enter values: " << endl;
for (int i = 0; i < row; i++)
arr2D[i] = new int[col];
```

```
for (int i = 0; i < row; i++)
{
    for (int j = 0; j < col; j++)
        cin >> arr2D[i][j];
}
return 0;
}
```

Declaration

Declaration

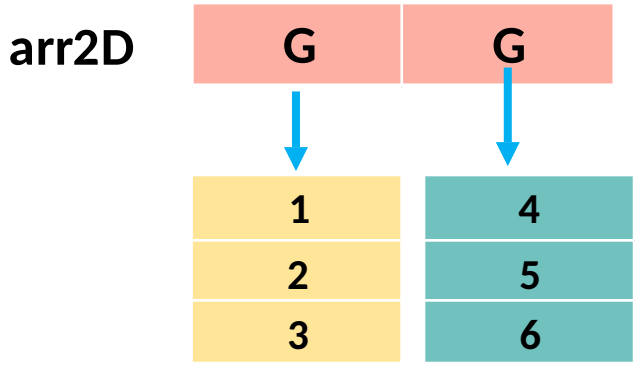
```
int size = 3;
int row, col;
cout << "Enter dimension 1 (# of rows:) " << endl;
cin >> row;
cout << "Enter dimension 2 (# of cols:) " << endl;
cin >> col;

int **arr2D;
arr2D = new int*[row];
cout << "Enter values: " << endl;
for (int i = 0; i < row; i++)
arr2D[i] = new int[col];
```



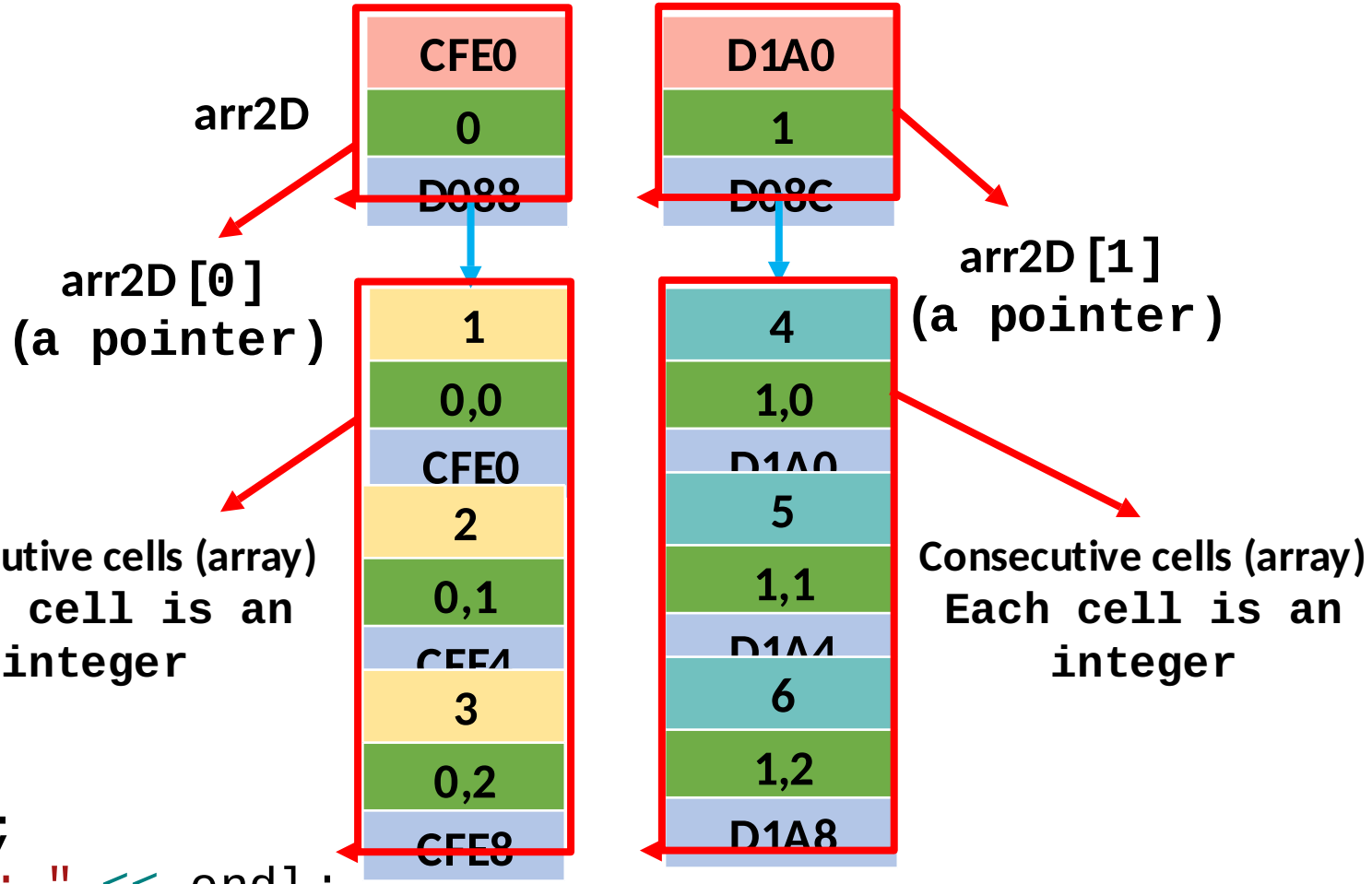

```
int **arr2D;
arr2D = new int*[row];
cout << "Enter values: " << endl;
for (int i = 0; i < row; i++)
     arr2D[i] = new int[col];
```

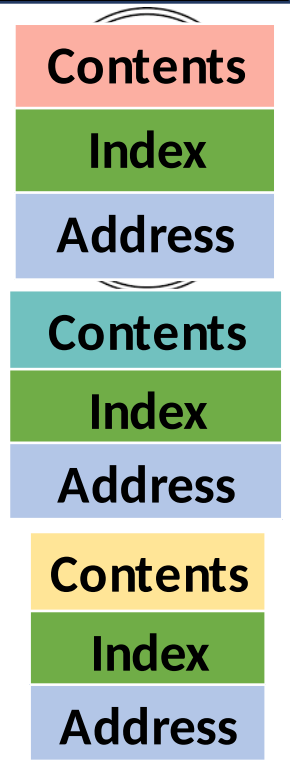
OR

Concise view

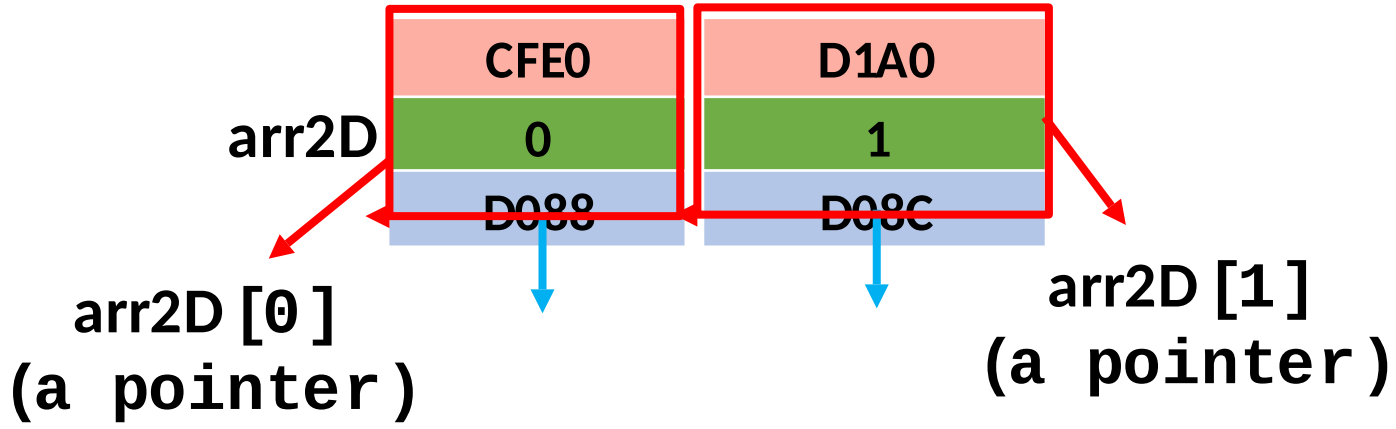


Detailed view



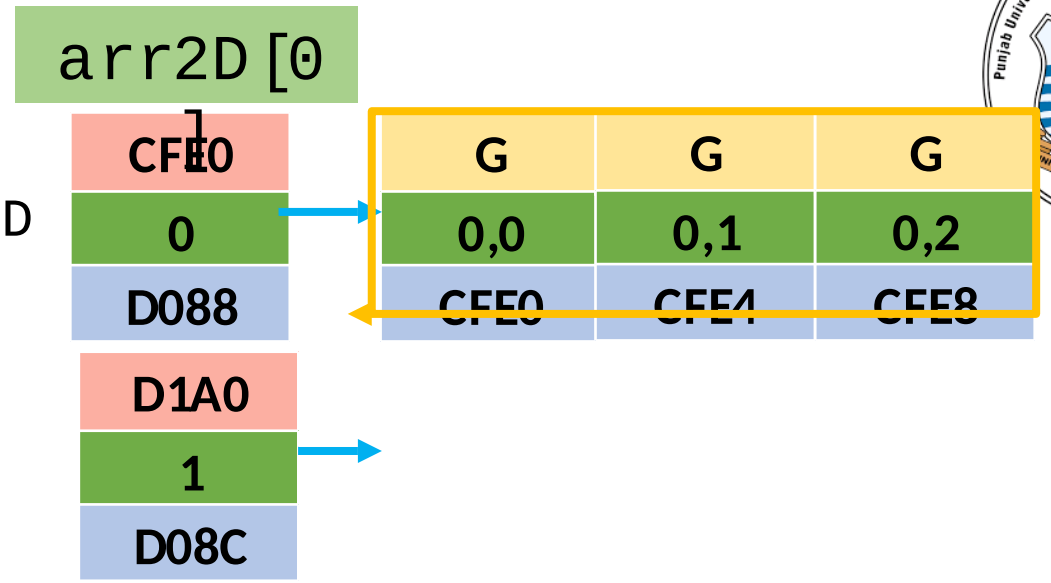



```
int **arr2D;
arr2D = new int*[row];
cout << "Enter values: " << endl;
for (int i = 0; i < row; i++)
     arr2D[i] = new int[col];
```




```
int **arr2D;
arr2D = new int*[row];   //row=2
```

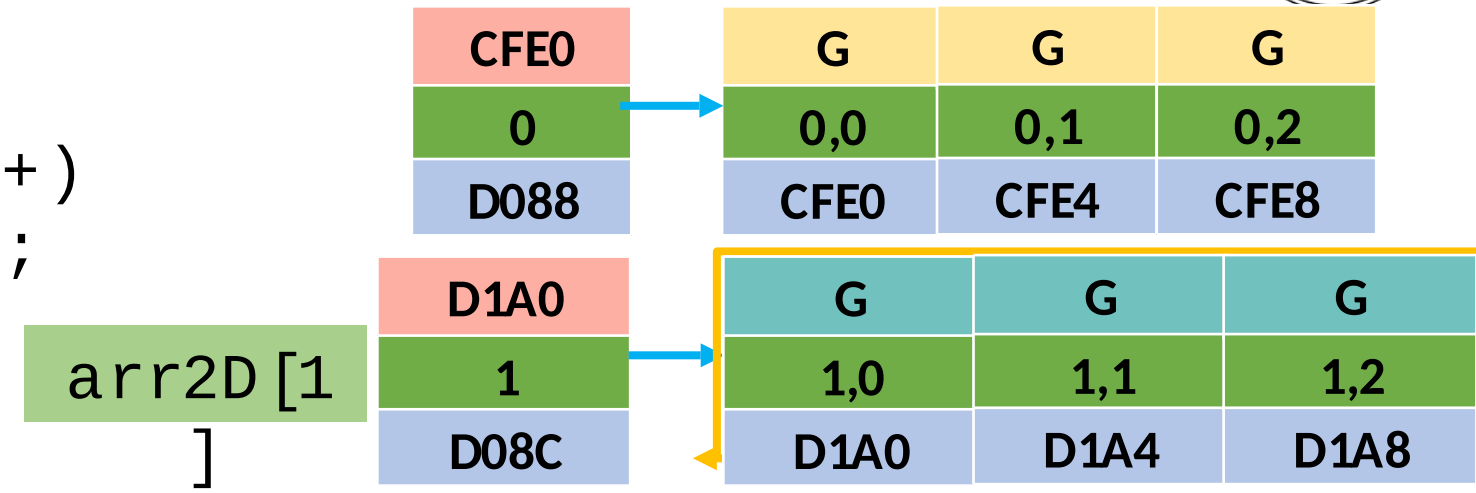
```
for (int i = 0; i < row; i++)
    arr2D[i] = new int[col];
```



i=0

```
arr2D[0] = new int[3];
```

```
for (int i = 0; i < row; i++)
    arr2D[i] = new int[col];
```



i=1

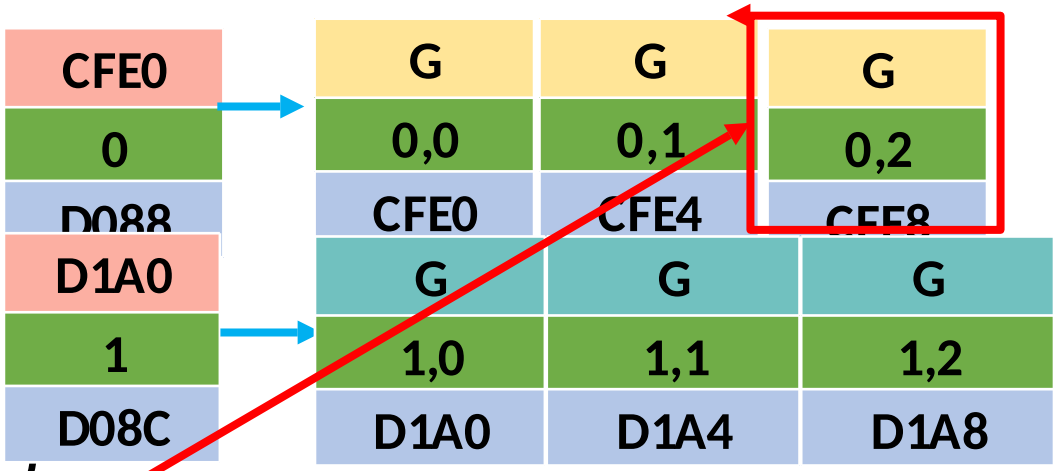
```
arr2D[1] = new int[3];
```

Taking input
(same procedure as studied for static 2D array)

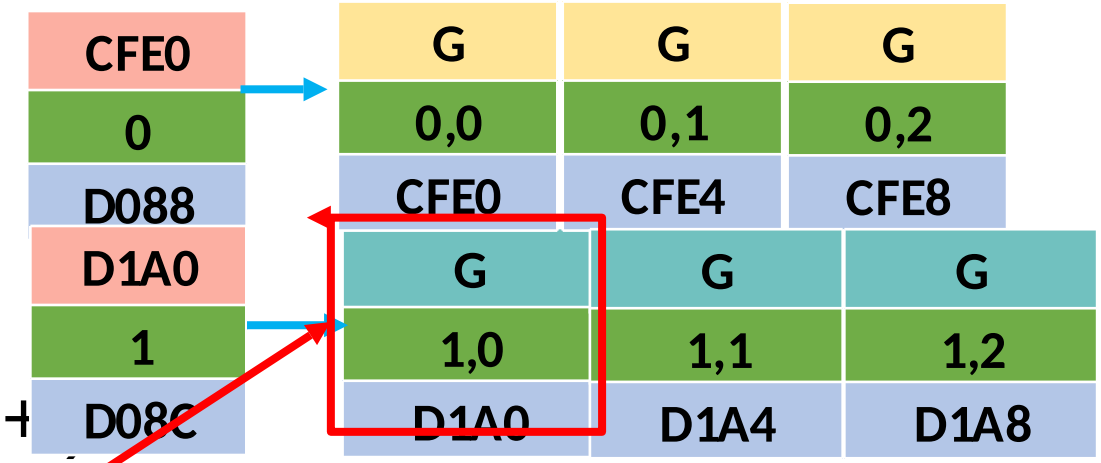
```
for (int i = 0; i < row; i++
{
  i=0

  for (int j = 0; j < col; j++)
      cin >> arr2D[i][j];
    j=0
    cin >> arr2D[0
}    ][0];
```

```
for (int i = 0; i < row; i++)
{
  i=0

  for (int j = 0; j < col; j++)
      cin >> arr2D[i][j];
    j=1
    cin >> arr2D[0
}    ][1];
```

```
for (int i = 0; i < row; i++)
{
  i=0

  for (int j = 0; j < col; j++)
      cin >> arr2D[i][j];
    j=2
    cin >> arr2D[0
}     ][2];
```



```
for (int i = 0; i < row; i++)
{
  i=1

  for (int j = 0; j < col; j++
      cin >> arr2D[i][j];
    j=0
    cin >> arr2D[1
}     ][0];
```



```
for (int i = 0; i < row; i++)
{
  i=1

  for (int j = 0; j < col; j++)
      cin >> arr2D[i][j];
    j=1
    cin >> arr2D[1
}    ][1];
```

```
for (int i = 0; i < row; i++)
{
  i=1
  for (int j = 0; j < col; j++)
      cin >> arr2D[i][j];
    j=2
    cin >> arr2D[1
}    ][2];
```

Deleting allocated memory

```
for (int i = 0; i < row
;i++)
    delete[] arr2D[i];
delete[] arr2D;
```

Deallocating each cell allocated using **new**

Deallocating array of pointers created!

Passing 2D array to a function

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

void show(int** arr, int x, int y)
{
    int i, j;
    for (i = 0; i < x; i++)
    {
        for (j = 0; j < y; j++)
        {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

```
int main()
{
    int n, m;
    cout << "Enter No. of rows: ";
    cin >> n;
    cout << "Enter No. of columns: ";
    cin >> m;
    int** A = new int* [n];
    for (int i = 0; i < n; i++)
    {
        A[i] = new int[m];
    }
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cin >> A[i][j];
        }
    }
    show(A, n, m);
    return 0;
}
```

Programming fundamentals

Lecture 23: *Stream input/output, Introduction to file handling, open a file for I/O, file modes*

Recap

- Dynamic array 1D
- Dynamic array 2D

Agenda

- Streams
- Input/output stream
- Intro to file handling
- Why files?
- Open a file
- Different modes of opening a file
- Opening a file for input
- Opening a file for output (with trunc and append mode)
- Close a file

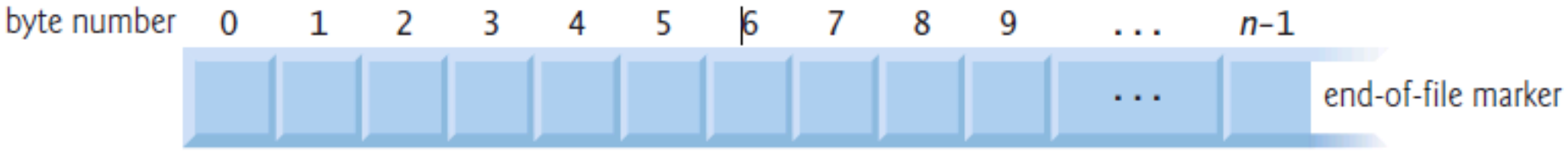
Stream

- C++ I/O occurs in streams, which are sequences of bytes.
- In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection, etc.) to main memory.

- In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection, etc.).

Why files?

- Storage of data in memory is temporary
- Files are used for persistent data storage
- Computers store files at secondary storage such has hard disk
- C+ views each file simply as a sequence of byters.
- Each file ends with an end of file marker



Contd..

- The bytes could represent characters, raw data, graphics images, digital speech, digital video or any other information an application may require.
- The system I/O mechanisms should transfer bytes from devices to memory (and vice versa) consistently and reliably.
- Such transfers often involve some mechanical motion, such as the rotation of a disk or a tape, or the typing of keystrokes at a keyboard.

Contd..

- The bytes could represent characters, raw data, graphics images, digital speech, digital video or any other information an application may require.
- The system I/O mechanisms should transfer bytes from devices to memory (and vice versa) consistently and reliably.
- Such transfers often involve some mechanical motion, such as the rotation of a disk or a tape, or the typing of keystrokes at a keyboard.

- The <iostream> header, which declares basic services required for all stream-I/O operations. The <iostream> header defines the cin, cout
- The <iomanip> header declares services useful for performing formatted I/O with socalled parameterized stream manipulators, such as setw and setprecision.
- The <fstream> header declares services for file processing.

ifstream

Input File Stream. The "i" stands for input and the "f" stands for file; so, a stream object from which data may be read.

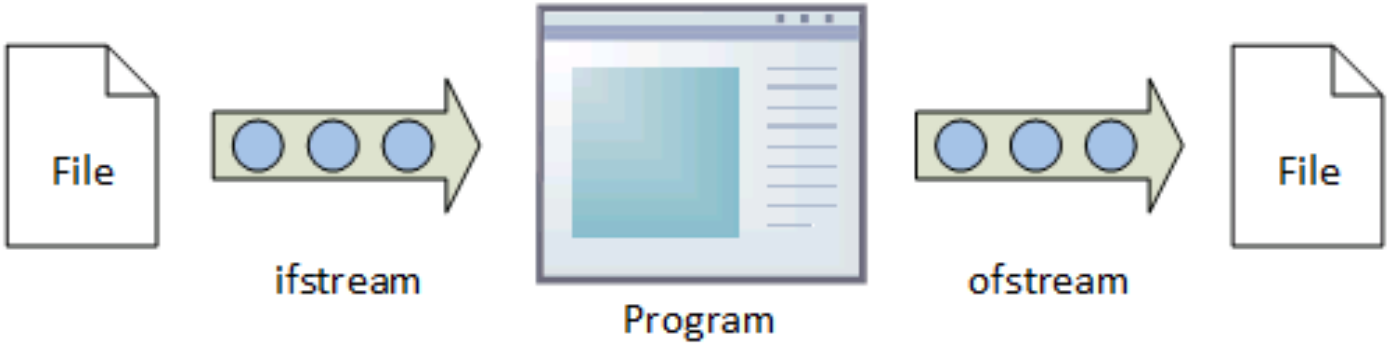
ofstream

Output File Stream. The "o" stands for output and the "f" stands for file; so, a stream object to which data may be written.

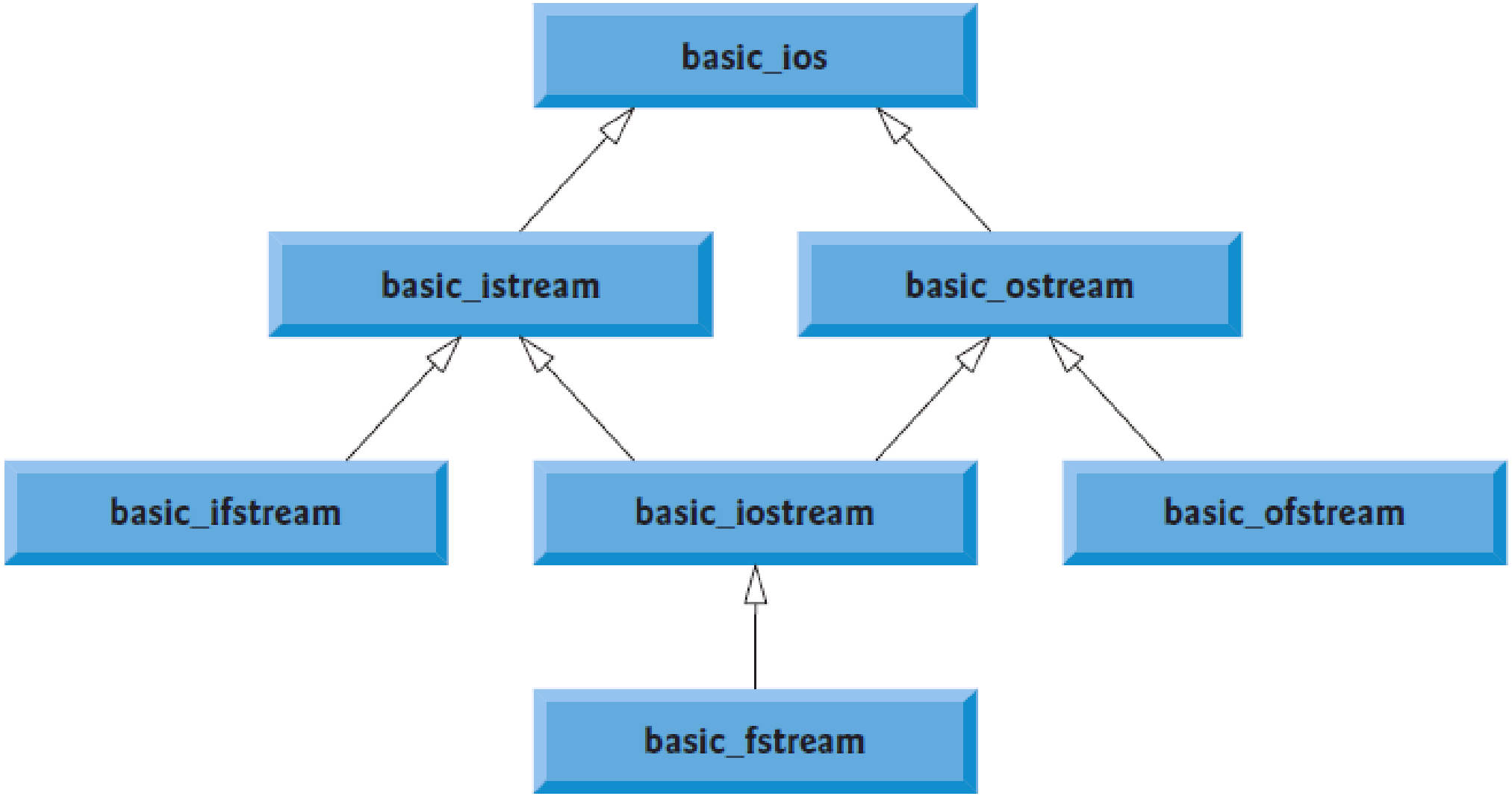
fstream

File Stream. The "f" stands for file but there is no indication of direction; so, a stream object that allows both reading and writing.

(b)



fstream has both ifstream and ofstream



File I/O

- To perform file processing, header ***<iostream> and <fstream>*** must be included.
- When a file is opened, an object is created and stream is associated with the object

For both Input/output

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{

  return 0;
}
```

For Input only

```
#include<iostream>
#include<ifstream>
using namespace std;
int main()
{

  return 0;
}
```

For Output only

```
#include<iostream>
#include<ofstream>
using namespace std;
int main()
{

  return 0;
}
```

Fstream objects

- Unlike the console stream objects that are ready for use as soon as the program starts, instances of the <fstream> classes must be instantiated and associated with a file before a program is able to use them.

Can use ***in*** only to read from file (Input)
You cannot write to a file using ***in***

ifstream in;

ofstream out;

Can use ***out*** only to read from file (Input)
You cannot write to a file using ***out***

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
    ifstream in;
    ofstream out;
    return 0;
}
```

Open a file

- We can open the file for input/output by ***creating an ifstream/ofstream/fstream object***.
- Two arguments are passed to the object's constructor—thefilename and the fileopen mode

File modes

Mode	Description
`ios::app`	*Append* all output to the end of the file.
`ios::ate`	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written *anywhere* in the file.
`ios::in`	Open a file for *input*.
`ios::out`	Open a file for *output*.
`ios::trunc`	*Discard* the file's contents (this also is the default action for `ios::out`).
`ios::binary`	Open a file for binary, i.e., *nontext*, input or output.

Opening a file for input

Open a file (using ifstream)

- Two arguments are passed to the object's constructor—thefilename and the fileopen mode

ifstream in("clients.txt", ios::in);

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
    ifstream in("clients.txt", ios::in);
    return 0;
}
```

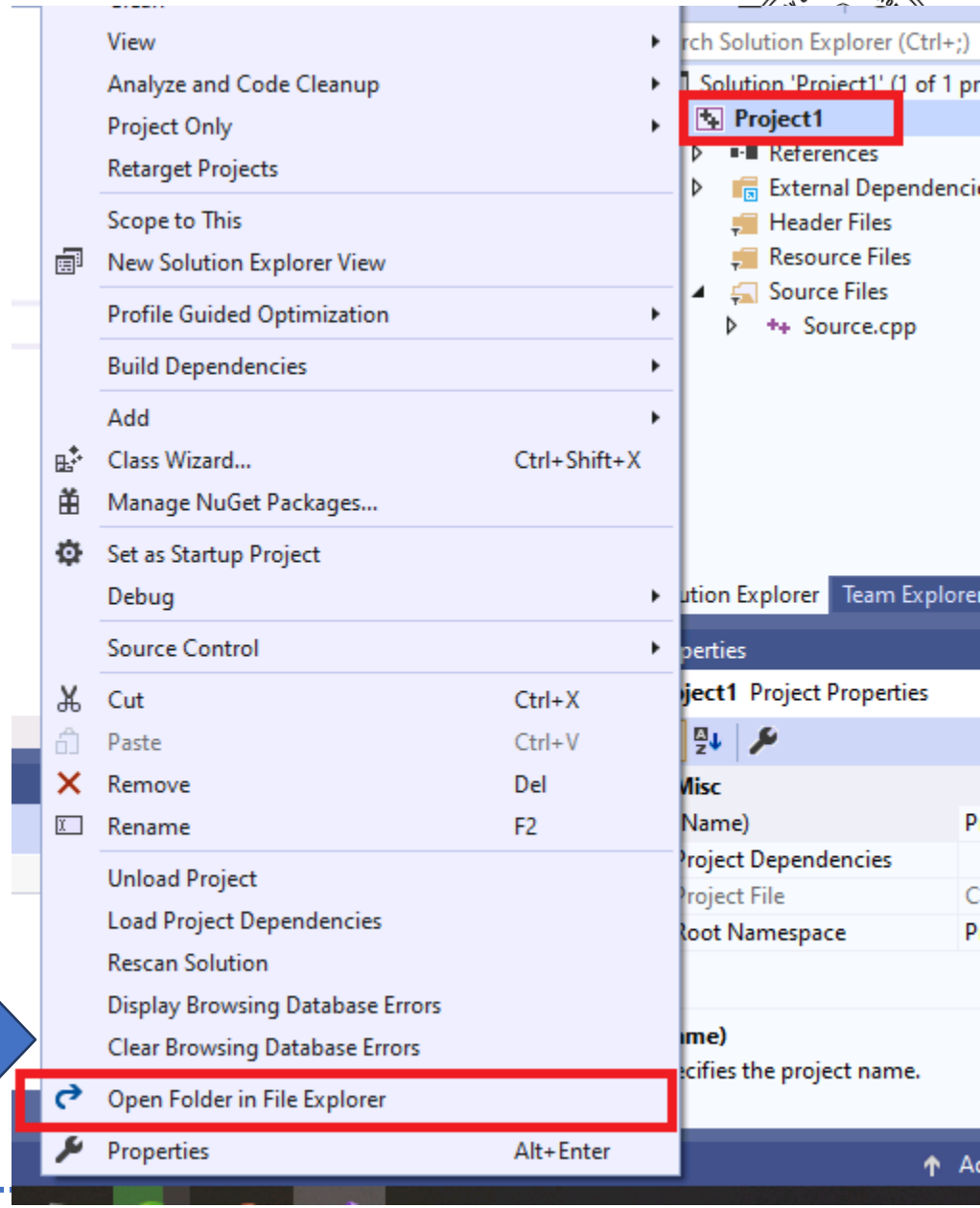
Before opening a file for input, the file MUST exist!

```
ifstream in( "clients.txt", ios::in );
```

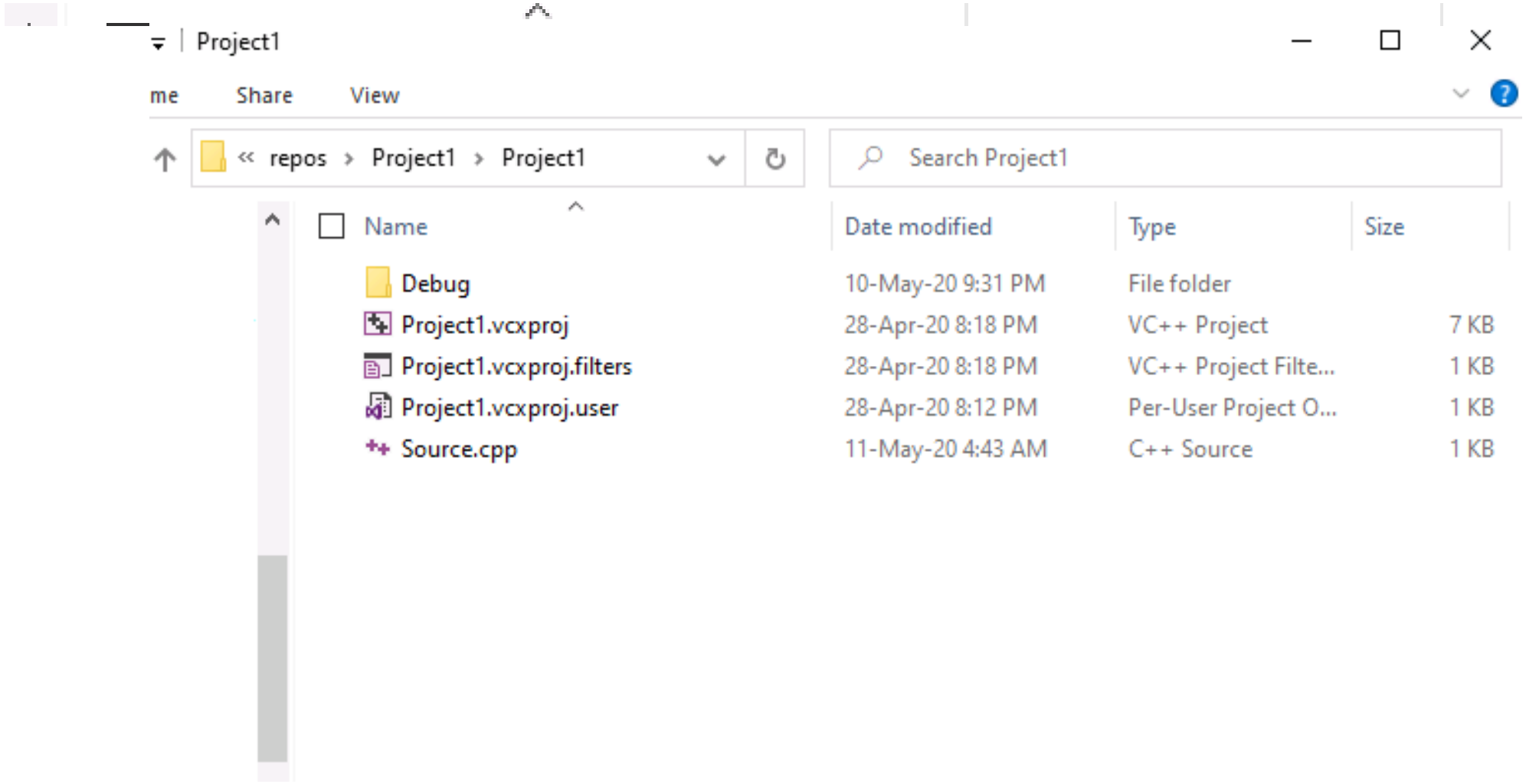
Here you specify the path of file! **i.e.** *path of location where the file resides*

If no path is specified, instead only file name I written, then the compiler assumes that the file resides in your current project directory (folder where your C++ current project is created on disk)

See this figure to open that folder. Right click on Project and then choose "Open Folder in File Explorer" from the drop down menu



C:\Users\Natalia\source\repos\Project1\Project1



Points to note!

- If you attempt to open a file for input that does not exist, then **while reading from the file** (which we will see later) will generate error!
- Make sure to open only those files for input that exist on your system!

Open a file (using fstream)

- The procedure and syntax will remain same as for ifstream!
- Two arguments are passed to the object's constructor—the filename and the fileopen mode

fstream f("clients.txt", ios::in);

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
    fstream f("clients.txt", ios::in);
    return 0;
}
```

Before opening a file for input, the file MUST exist!

Open a file (for output)

- Two arguments are passed to the object's constructor—thefilename and the fileopen mode

ofstream out("clients.txt", ios::out);

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
   ifstream in;
   ofstream out("clients.txt", ios::out);
   return 0;
}
```

Opening a file for output

Open a file (using ofstream)

- Two arguments are passed to the object's constructor—thefilename and the fileopen mode

ofstream out("clients.txt", ios::out);

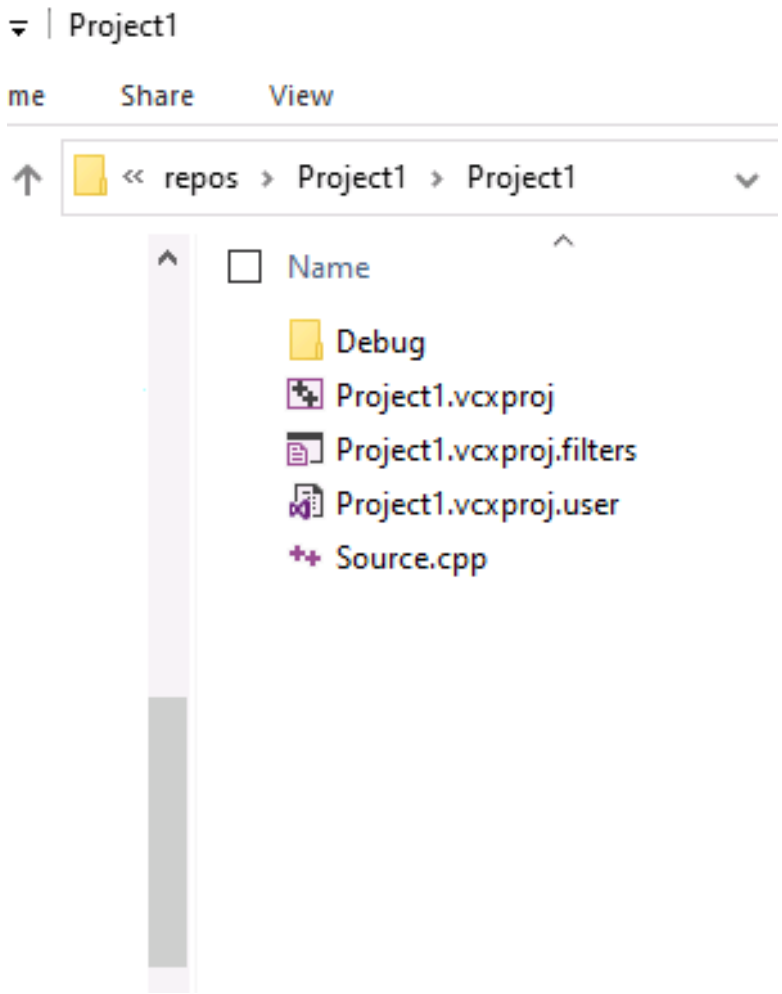
```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
    ofstream out("clients.txt", ios::out)
    return 0;
}
```

Before opening a file for out, there is no restriction that the file must exist!

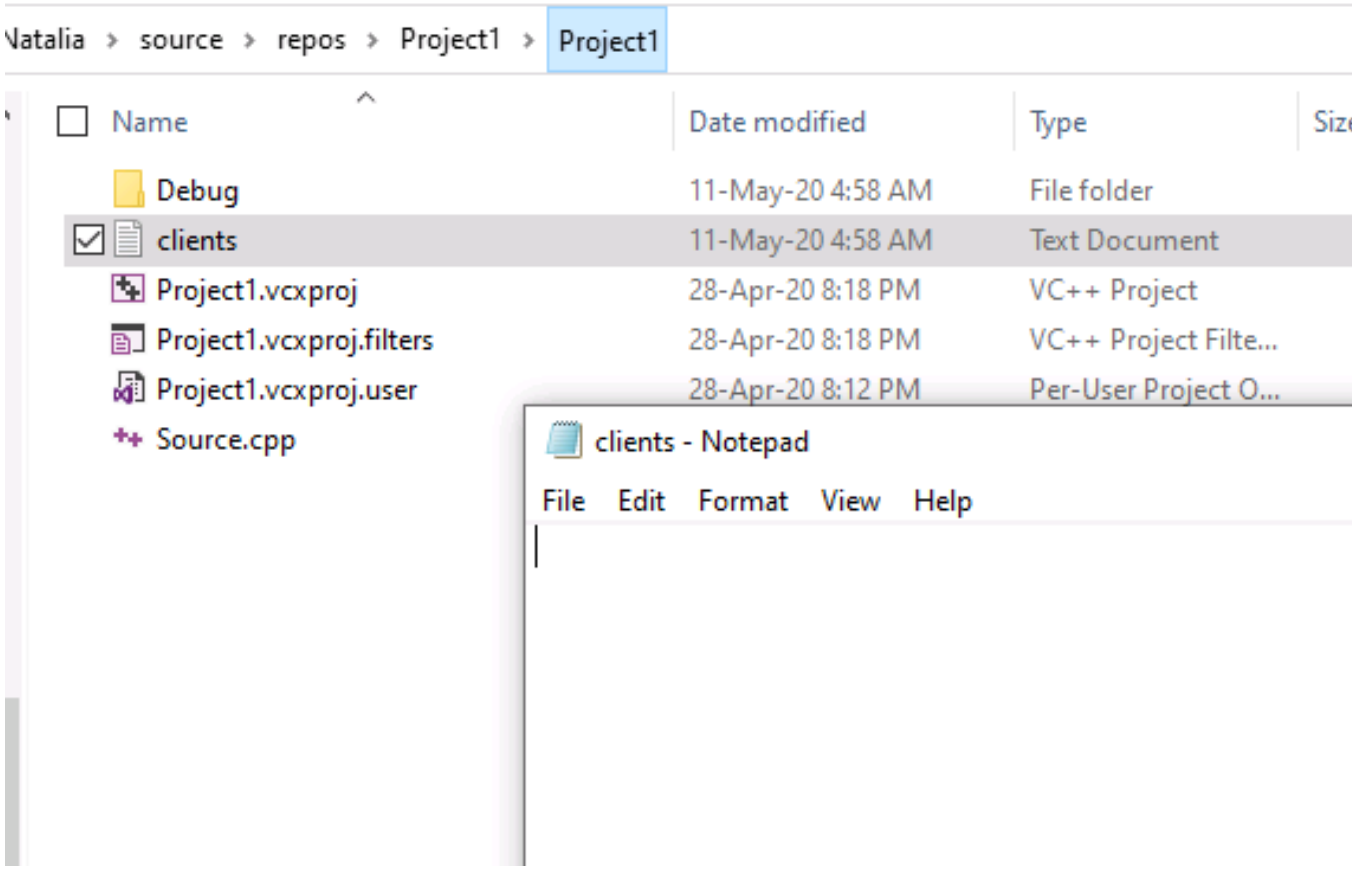
1. ***If the file exists, it will be opened in truncate mode (by default;unless you specify other mode)***
2. ***If the file does not exist, then it will be created and then opened in truncate mode.***

File opened (file does not exist!)

Before running code



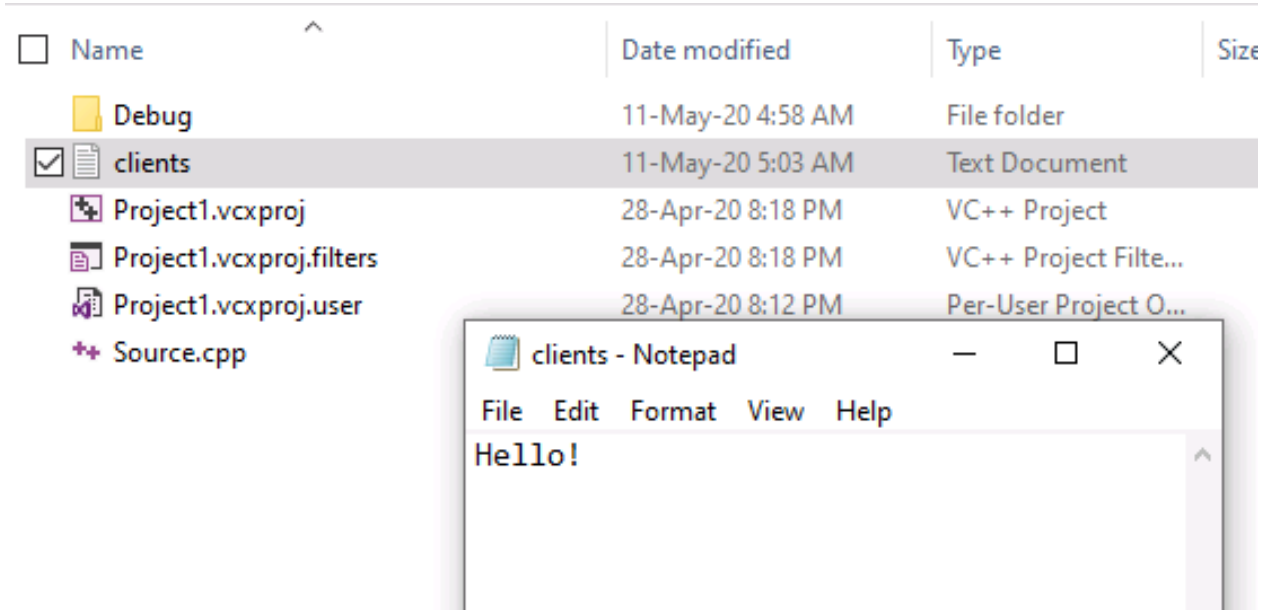
After running code



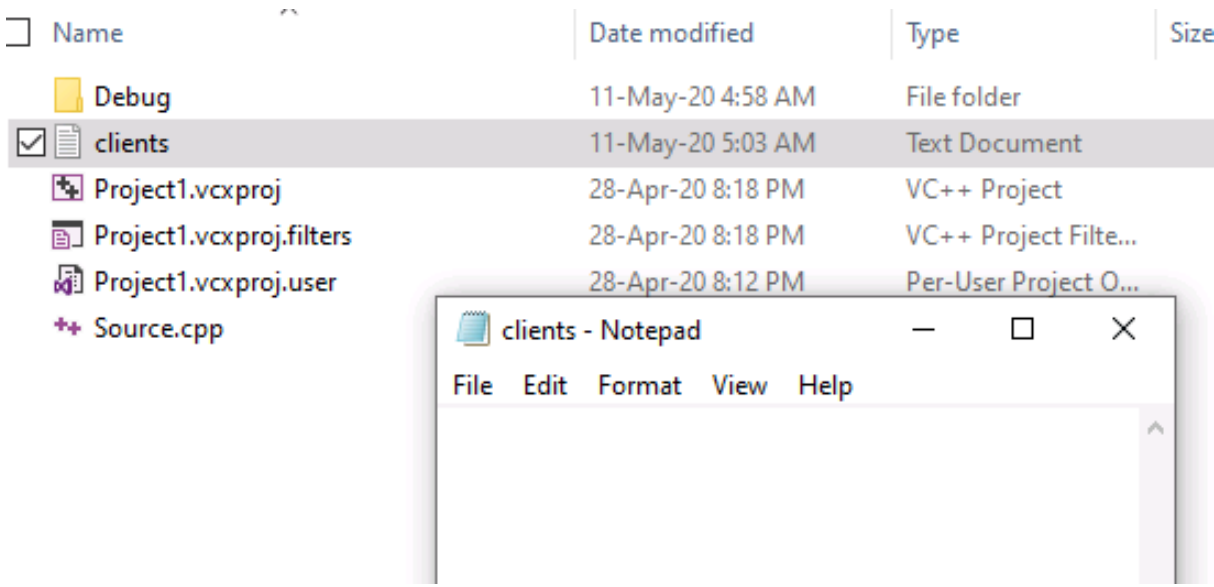
File opened (file exists with some data!)

The data got truncated!

Before running code



After running code



Suppose we want to not truncate the data..instead we want to append the data to existing data!

We need to use ios::app mode!

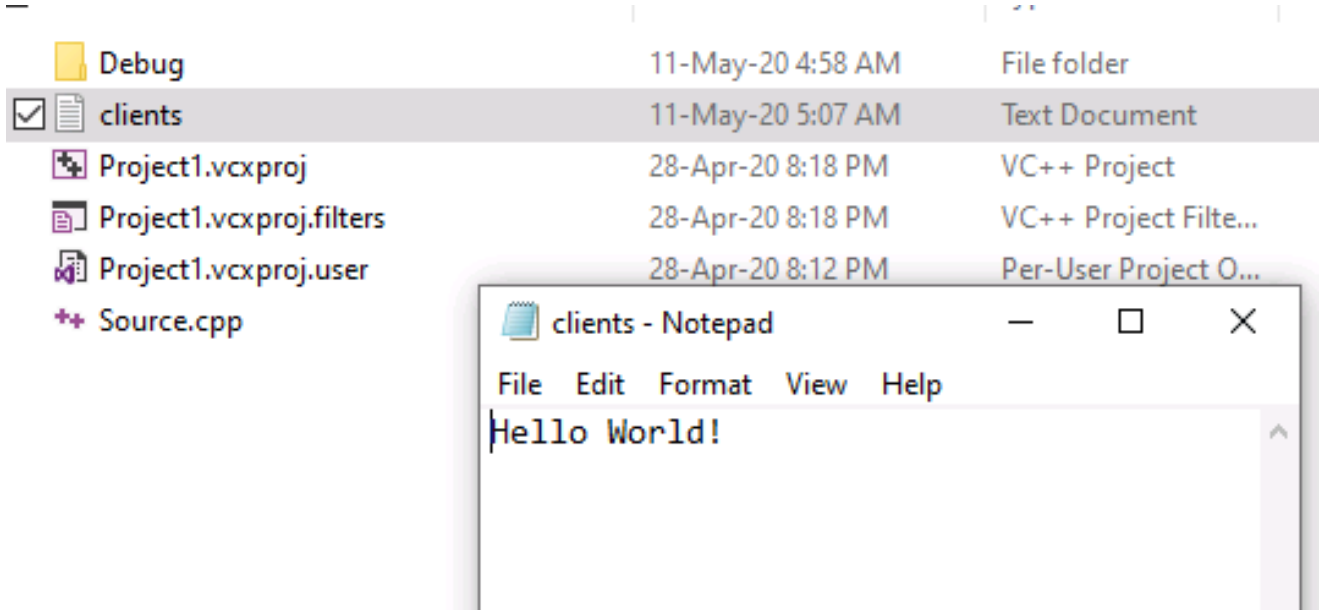
C++ allows you to use more than one file modes

- You need to separate the files modes by a “ |” sign!

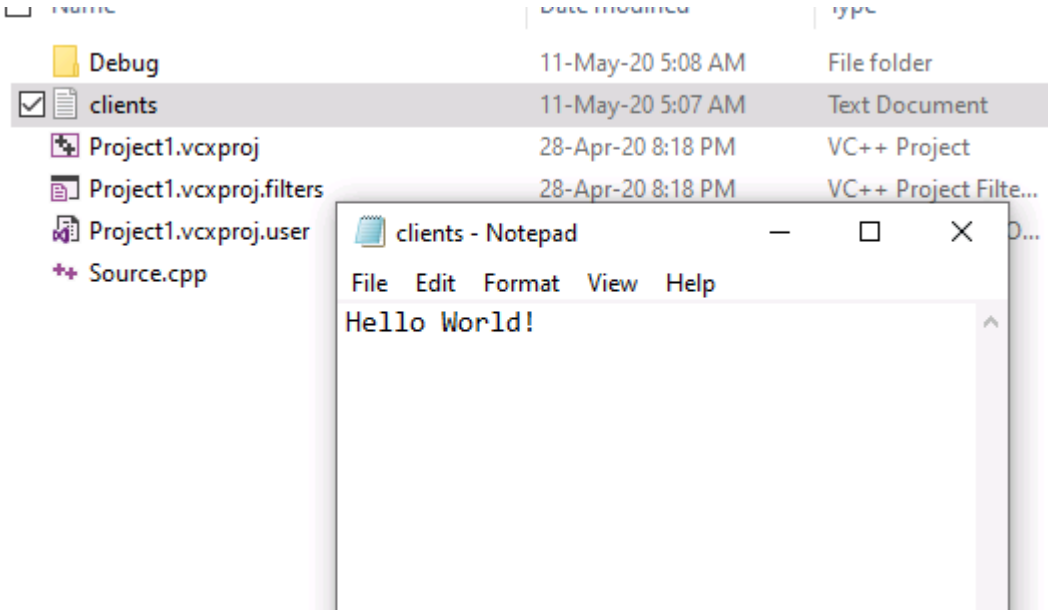
```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   ofstream out("clients.txt", ios::out | ios::app
   );
   return 0;
}
```

The data is not truncated! The file pointer will point after the last character "!" i.e. at EOF

Before running code



After running code



Any text you write in file will go at the end of existing text (Hello World!)

Closing a file

- It is mandatory and preferred to ALWAYS close a file after using it (either for Input or for Output)

outClientFile.close();

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream out("clients.txt", ios
    ::out | ios::app);
    out.close();
    return 0;
}
```

Open function

- We just covered how to open a file for I/O
- The method that we covered is called “constructor method”
- There is also another way to open a file. That is done using **open** function..

Open a file using **open function**

Same as:

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream out("clients.txt", ios::out | ios
    ::app);
    out.close();
    return 0;
}
```

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream out;
    out.open("clients.txt", ios::out  | ios::app);
    out.close();
    return 0;
}
```

Same goes for opening a file for input!

End

Programming fundamentals

Lecture 24: Read/write to a file

Recap

- Streams
- Input/output stream
- Intro to file handling
- Why files?
- Open a file
- Different modes of opening a file
- Opening a file for input
- Opening a file for output (with trunc and append mode)
- Close a file

Stream

- C++ I/O occurs in streams, which are sequences of bytes.
- In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection, etc.) to main memory.

- In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection, etc.).

ifstream

Input File Stream. The "i" stands for input and the "f" stands for file; so, a stream object from which data may be read.

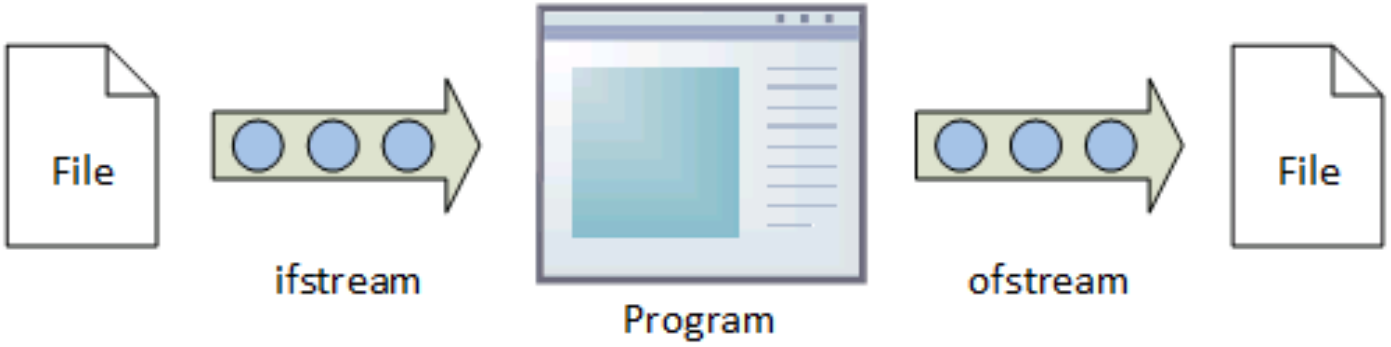
ofstream

Output File Stream. The "o" stands for output and the "f" stands for file; so, a stream object to which data may be written.

fstream

File Stream. The "f" stands for file but there is no indication of direction; so, a stream object that allows both reading and writing.

(b)



For both Input/output

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{

  return 0;
}
```

For Input only

```
#include<iostream>
#include<ifstream>
using namespace std;
int main()
{

  return 0;
}
```

For Output only

```
#include<iostream>
#include<ofstream>
using namespace std;
int main()
{

  return 0;
}
```

File modes

Mode	Description
`ios::app`	*Append* all output to the end of the file.
`ios::ate`	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written *anywhere* in the file.
`ios::in`	Open a file for *input*.
`ios::out`	Open a file for *output*.
`ios::trunc`	*Discard* the file's contents (this also is the default action for `ios::out`).
`ios::binary`	Open a file for binary, i.e., *nontext*, input or output.

Opening a file for input

Open a file (using ifstream)

- Two arguments are passed to the object's constructor—thefilename and the fileopen mode

ifstream in("clients.txt", ios::in);

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
    ifstream in("clients.txt", ios::in);
    return 0;
}
```

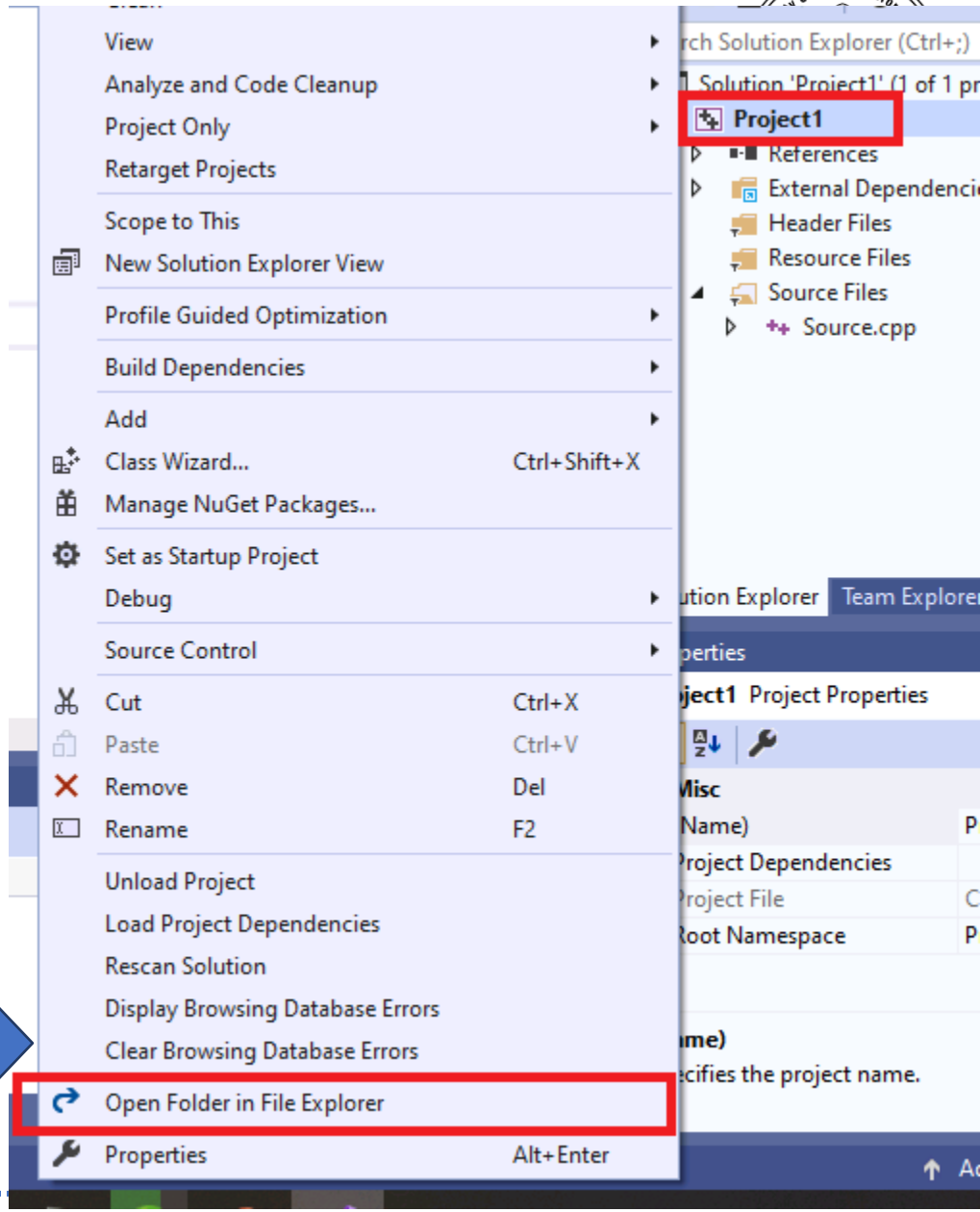
Before opening a file for input, the file MUST exist!

```
ifstream in( "clients.txt", ios::in );
```

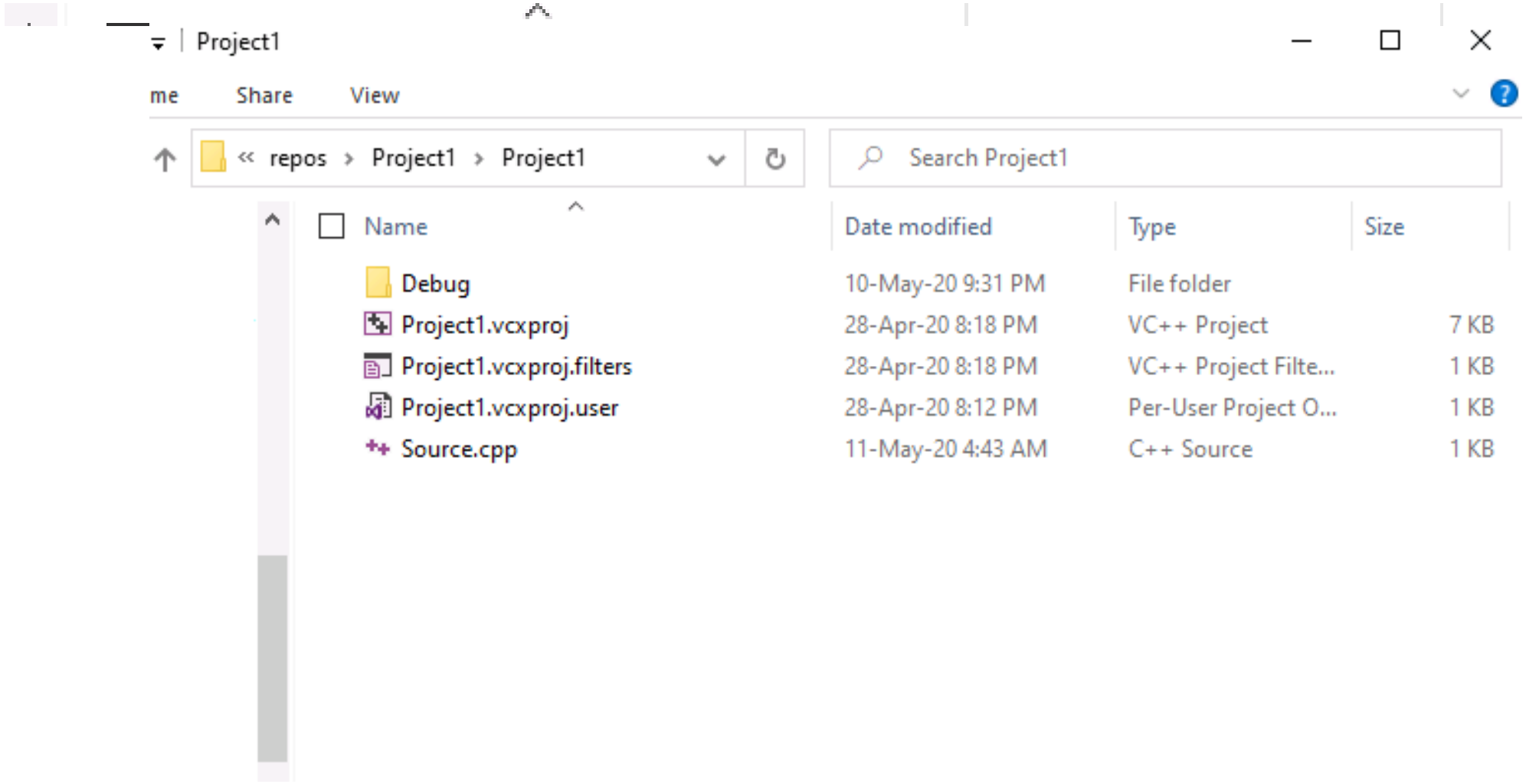
Here you specify the path of file! **i.e.** *path of location where the file resides*

If no path is specified, instead only file name I written, then the compiler assumes that the file resides in your current project directory (folder where your C++ current project is created on disk)

See this figure to open that folder. Right click on Project and then choose "Open Folder in File Explorer" from the drop down menu



C:\Users\Natalia\source\repos\Project1\Project1



Closing a file

- It is mandatory and preferred to ALWAYS close a file after using it (either for Input or for Output)

outClientFile.close();

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream out("clients.txt", ios
    ::out | ios::app);
    out.close();
    return 0;
}
```

Open function

- We just covered how to open a file for I/O
- The method that we covered is called “constructor method”
- There is also another way to open a file. That is done using **open** function..

Open a file using **open function**

Same as:

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream out("clients.txt", ios::out | ios
    ::app);
    out.close();
    return 0;
}
```

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream out;
    out.open("clients.txt", ios::out  | ios::app);
    out.close();
    return 0;
}
```

Same goes for opening a file for input!

Agenda

- Read from a file
- Write to a file

Default modes for ifstream,ofstream,fstream

What if we don’t specify the second parameter while opening a file

- Each one of the open() member functions of the classes ofstream , ifstream and fstream has a default mode that is used if the file is opened without a second argument:

ofstream ios::out
ifstream ios::in
fstream ios::in | ios::out

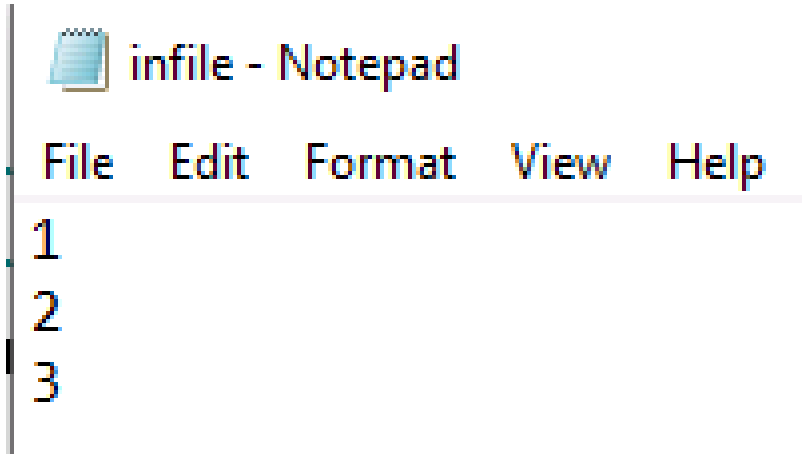
Read from file

Read from the file

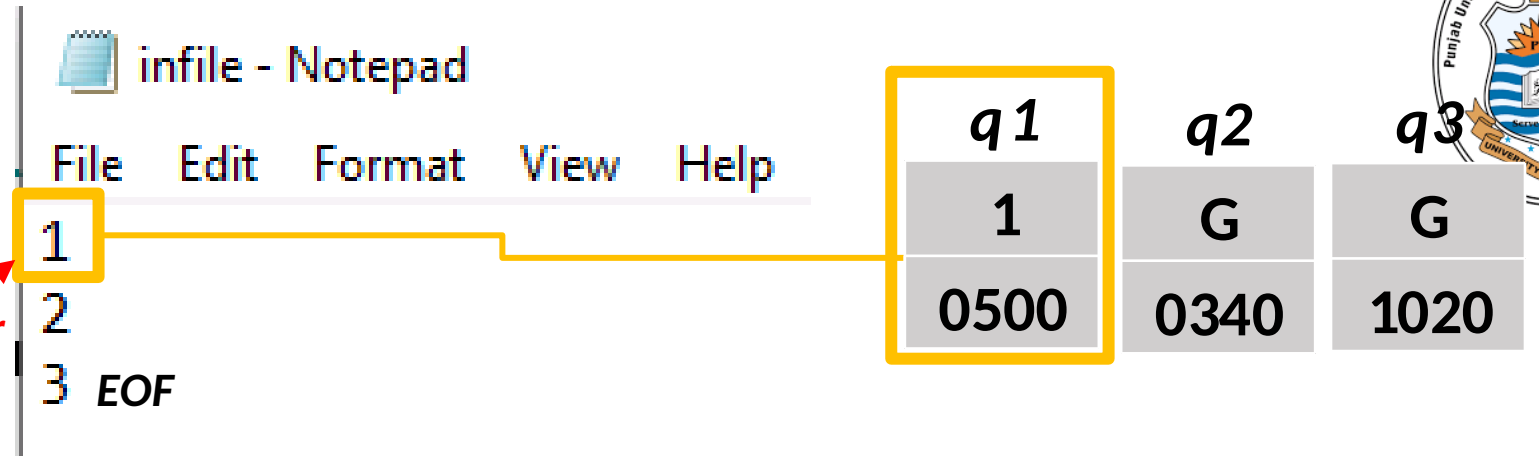
- The file is organized in memory as a sequence (stream) of bytes!

1	\n	2	\n	3	EoF	..	
0501	0502	0503	0504	0505	0506	..	

File pointer



Read from the file

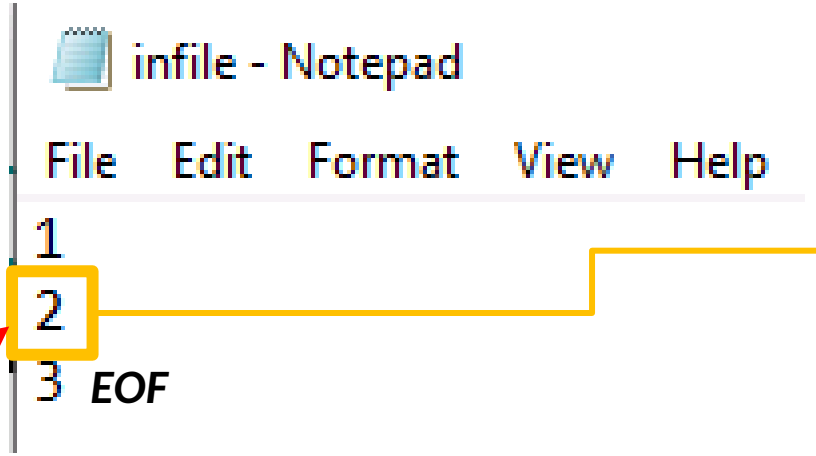



```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    int q1, q2, q3;
    fstream f;
    f.open("infile.txt");
    f >> q1; // just like how cin is used to
    read
                 // a number from the keyboard
    f >> q2;
    f >> q3;
    cout << q1 << " " << q2 << " " << q3 <<
    endl;
    f.close();
}
```

Character read from file and then cout to the screent

- *There is always a file pointer associated with the file.*
- *This pointer points at the start of the first initially.*
- *Once we read a character, the file pointer moves one character ahead automatically.*

Contd



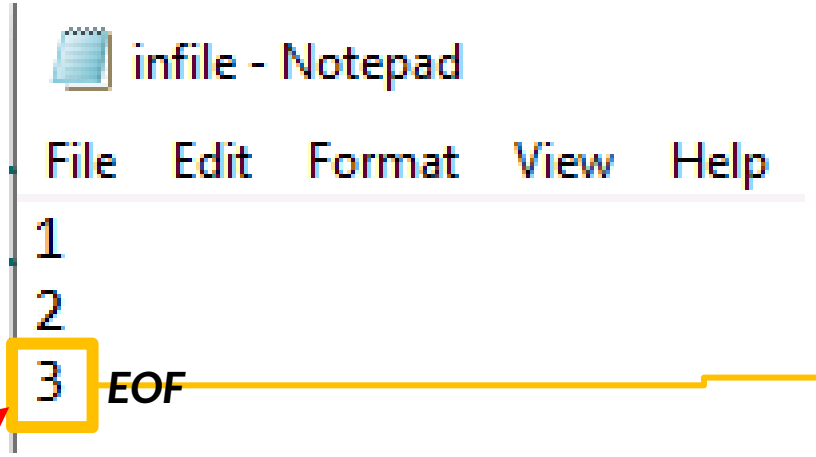
q1	q2	q3
1	2	G
0500	0340	1020

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    int q1, q2, q3;
    fstream f;
    f.open("infile.txt");
    f >> q1; // just like how cin is used to
    read
                 // a number from the keyboard
    f >> q2;
    f >> q3;
    cout << q1 << " " << q2 << " " << q3 <<
    endl;
    f.close();
}
```

Character read from file and then cout to the screent

Just like in cin, filestream also takes space and newline as a terminating character..

Contd



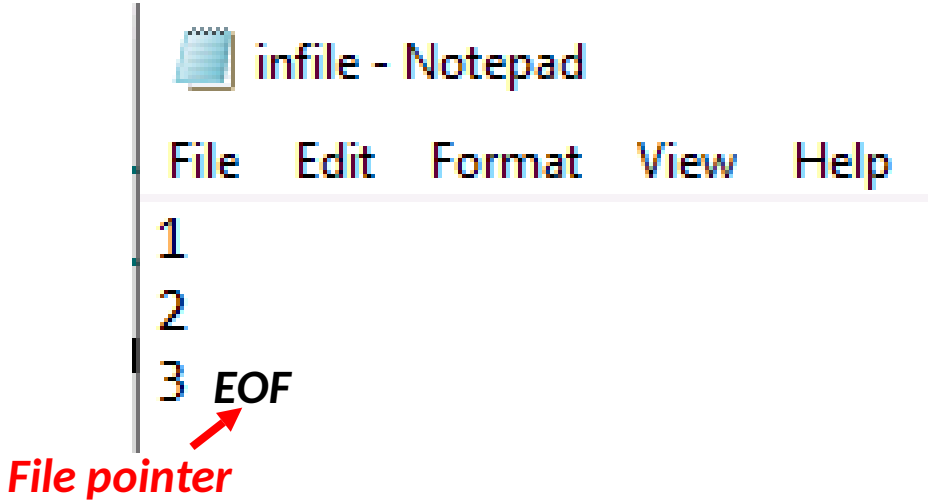
q1	q2	q3
1	2	3
0500	0340	1020

Character read from file and then cout to the screent

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    int q1, q2, q3;
    fstream f;
    f.open("infile.txt");
    f >> q1; // just like how cin is used to
    read
                 // a number from the keyboard
    f >> q2;
    f >> q3;
    cout << q1 << " " << q2 << " " << q3 <<
    endl;
    f.close();
}
```

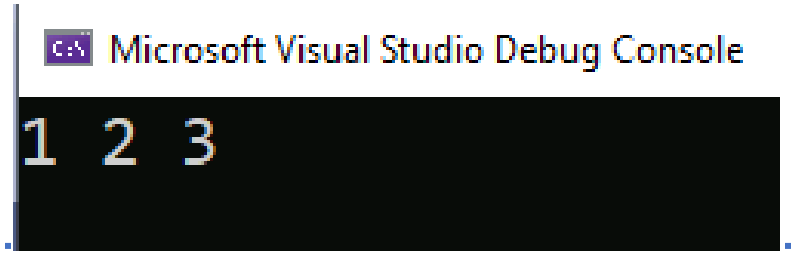
Just like in cin, filestream also takes space and newline as a terminating character..

Contd




```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    int q1, q2, q3;
    fstream f;
    f.open("infile.txt");
    f >> q1; // just like how cin is used to
    read
                 // a number from the keyboard
    f >> q2;
    f >> q3;
    cout << q1 << " " << q2 << " " << q3 <<
    endl;
    f.close();
}
```

Just like in cin, filestream also takes space and newline as a terminating character..



Write to a file

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    int marks=85;
    fstream f;
    f.open("outfile.txt",ios::out);
    f <<"My marks are:"; // just like
    how cout is used to write
                    // a number to the
    console
    f << endl;
    f << marks;
    f.close();
}
```

Sometimes fstream wont create an output file without second argument. Therefore, it is ideal to always mention mode

pos › Project1 › Project1

Search Project1

Name	Date modified
Debug	16-May-20 11:01 PM
infile	16-May-20 9:40 PM
outfile	16-May-20 11:01 PM
Project1.vcxproj	28-Apr-20 8:18 PM
Project1.vcxproj.filters	28-Apr-20 8:18 PM
Project1.vcxproj.user	28-Apr-20 8:12 PM
Source.cpp	16-May-20 11:01 PM

outfile - Notepad

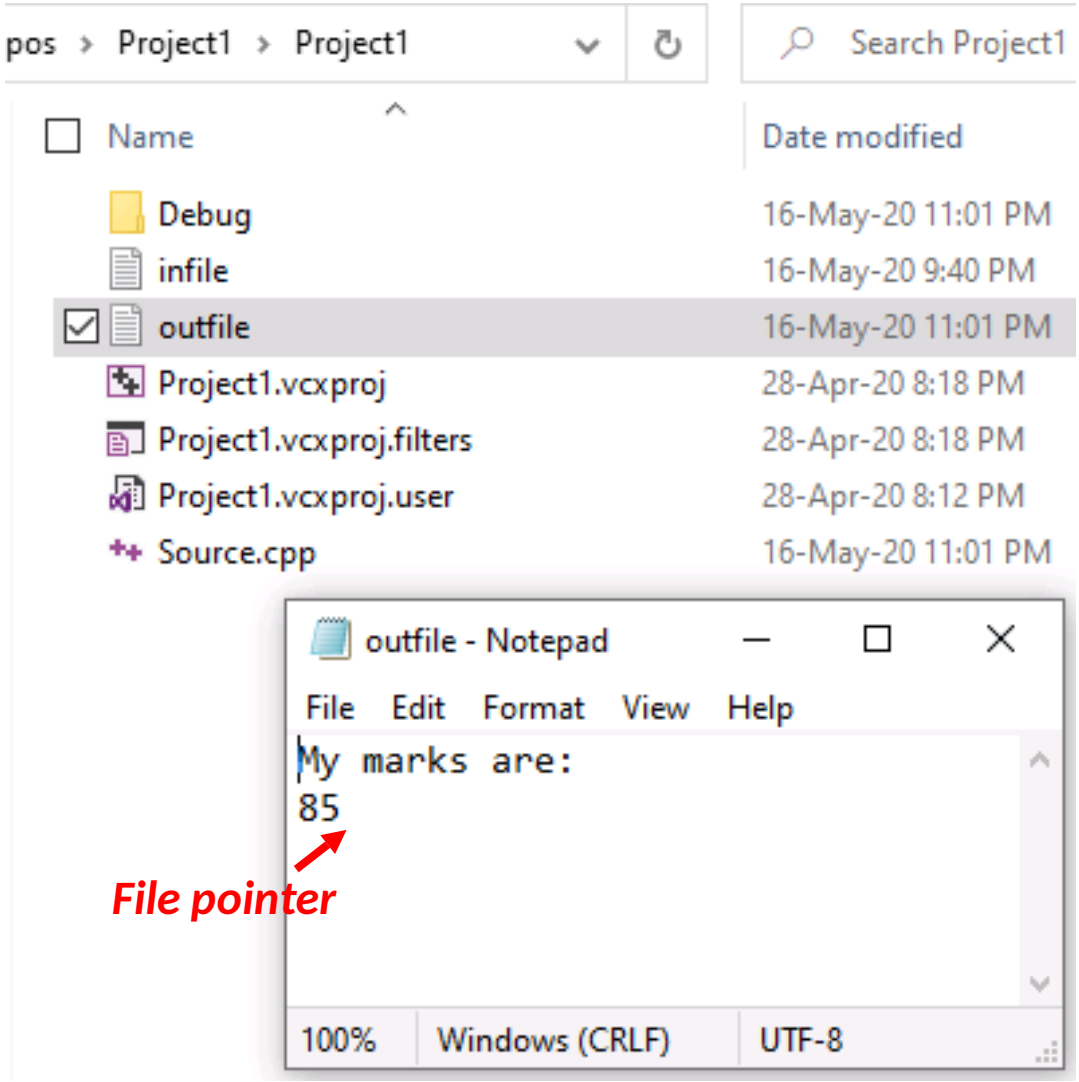
File Edit Format View Help

My marks are:
85 ***EOF***

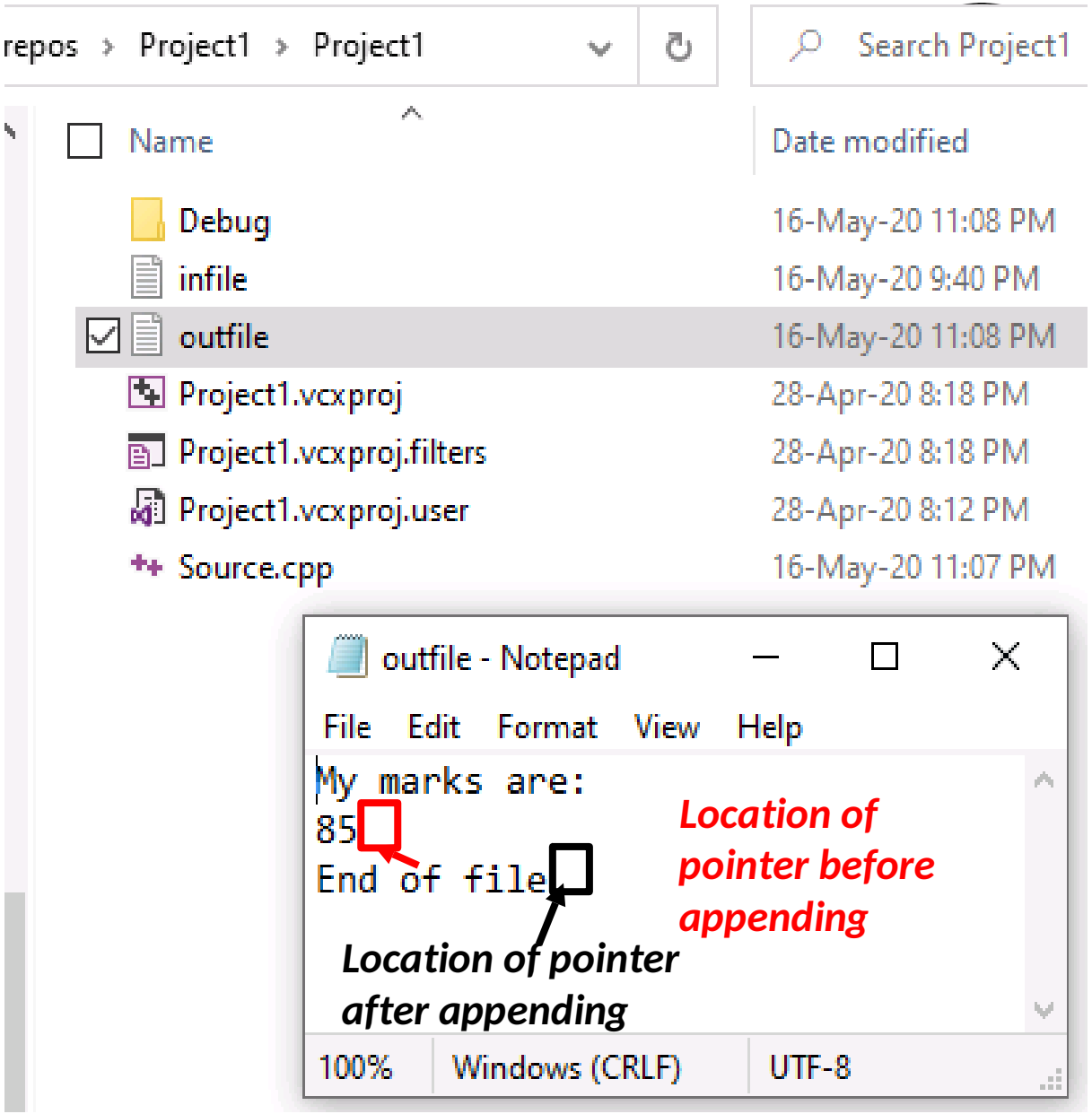
File pointer

100% Windows (CRLF) UTF-8

Append to the same file


pos › Project1 › Project1
Search Project1
Name
Date modified
Debug
16-May-20 11:01 PM
infile
16-May-20 9:40 PM
outfile
16-May-20 11:01 PM
Project1.vcxproj
28-Apr-20 8:18 PM
Project1.vcxproj.filters
28-Apr-20 8:18 PM
Project1.vcxproj.user
28-Apr-20 8:12 PM
Source.cpp
16-May-20 11:01 PM
outfile - Notepad
File Edit Format View Help
My marks are:
85
100%
Windows (CRLF)
UTF-8
File pointer

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    int marks=85;
    fstream f;
    f.open("outfile.txt",ios::app);
    f << endl;
    f << "End of file";
    f.close();
}
```



Read/write from file

Reads three numbers from the file, sums the numbers, and writes the sum to the file outfile.txt

epos › Project1 › Project1

Search Project1

Name	Date modified	Type	Size
Debug	16-May-20 11:08 PM	File folder	
infile	16-May-20 9:40 PM	Text Document	1 KB
outfile	16-May-20 11:08 PM	Text Document	1 KB
Project1.vcxproj	28-Apr-20 8:18 PM	VC++ Project	7 KB
Project1.vcxproj.filters	28-Apr-20 8:18 PM	VC++ Project Filte...	1 KB
Project1.vcxproj.user	28-Apr-20 8:12 PM	Per-User Project O...	1 KB
Source.cpp	16-May-20 11:19 PM	C++ Source	1 KB

infile - Notepad

File Edit Format View Help

```
1
2
3
```

100% Windows (CRLF) UTF-8

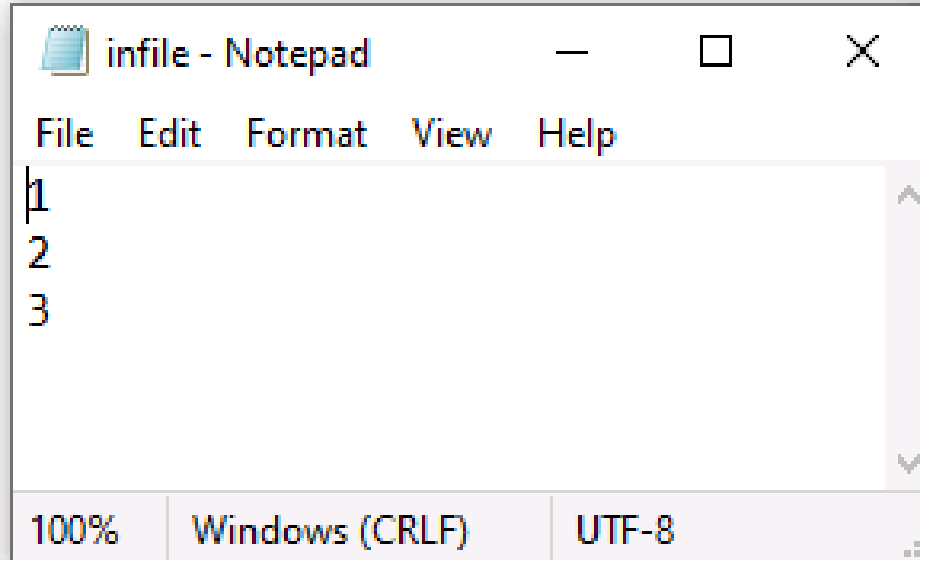
outfile - Notepad

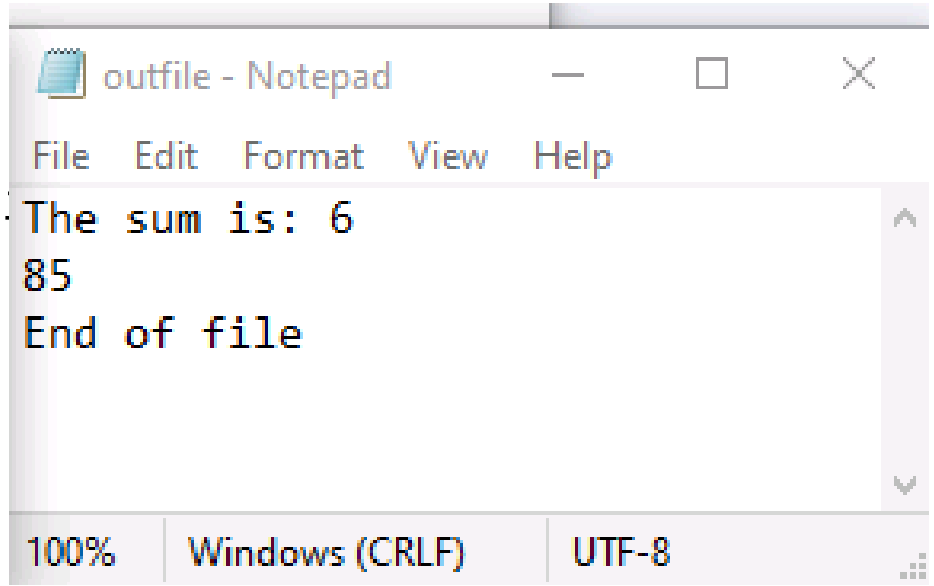
File Edit Format View Help

```
My marks are:
85
End of file
```

100% Windows (CRLF) UTF-8

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    fstream f1,f2;
    f1.open("infile.txt",ios::in|ios::out);
    f2.open("outfile.txt", ios::in | ios::out);
    int first, second, third;
    f1 >> first >> second >> third;
    f2 << "The sum is: " << (first + second + th
     << endl;
    f1.close();
    f2.close();
}
```





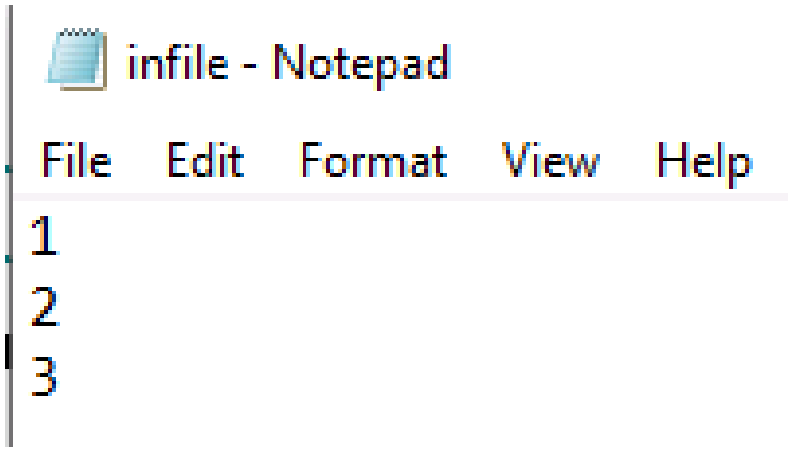
What if we open fstream with both in and out mode

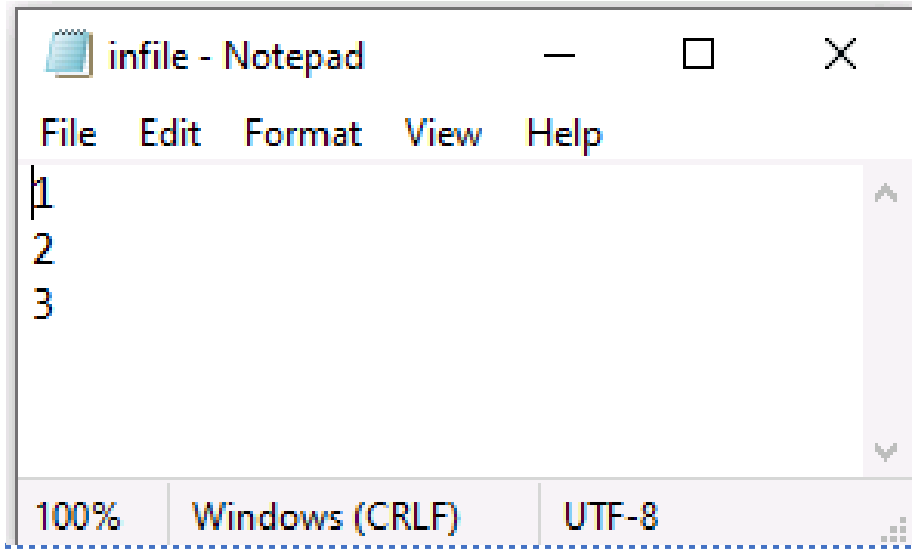
The fstream object is associated with the operation that is performed first!

Input is performed first, (then you cannot perform output with this object afterwards)

Pointer points at the start for reading data

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    fstream f1, f2;
    f1.open("infile.txt", ios::in | ios
::out);
    int first, second, third;
    f1 >> first >> second >> third;
    cout << first << second << third <<
endl;
    f1 << "Finished reading data";
    f1.close();
}
```

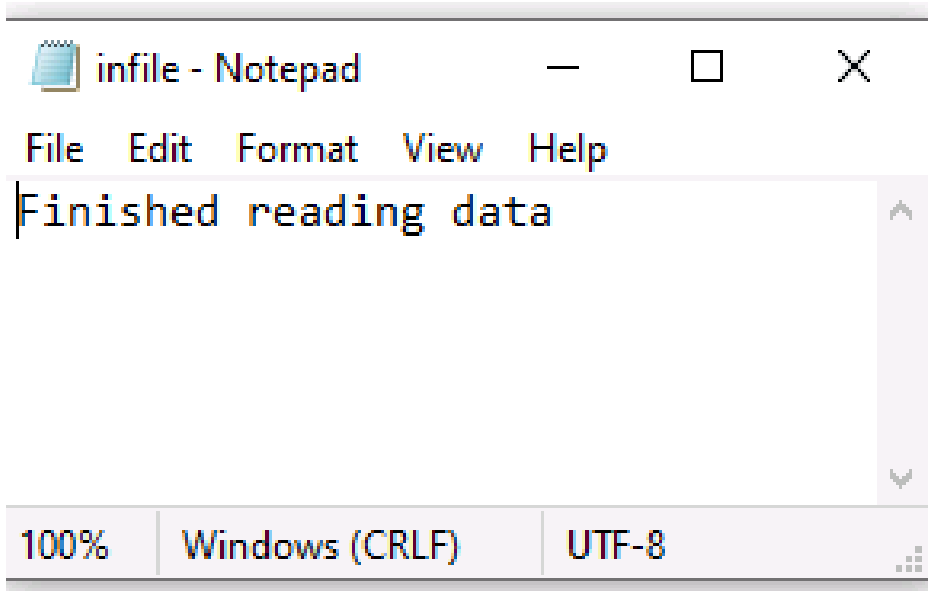




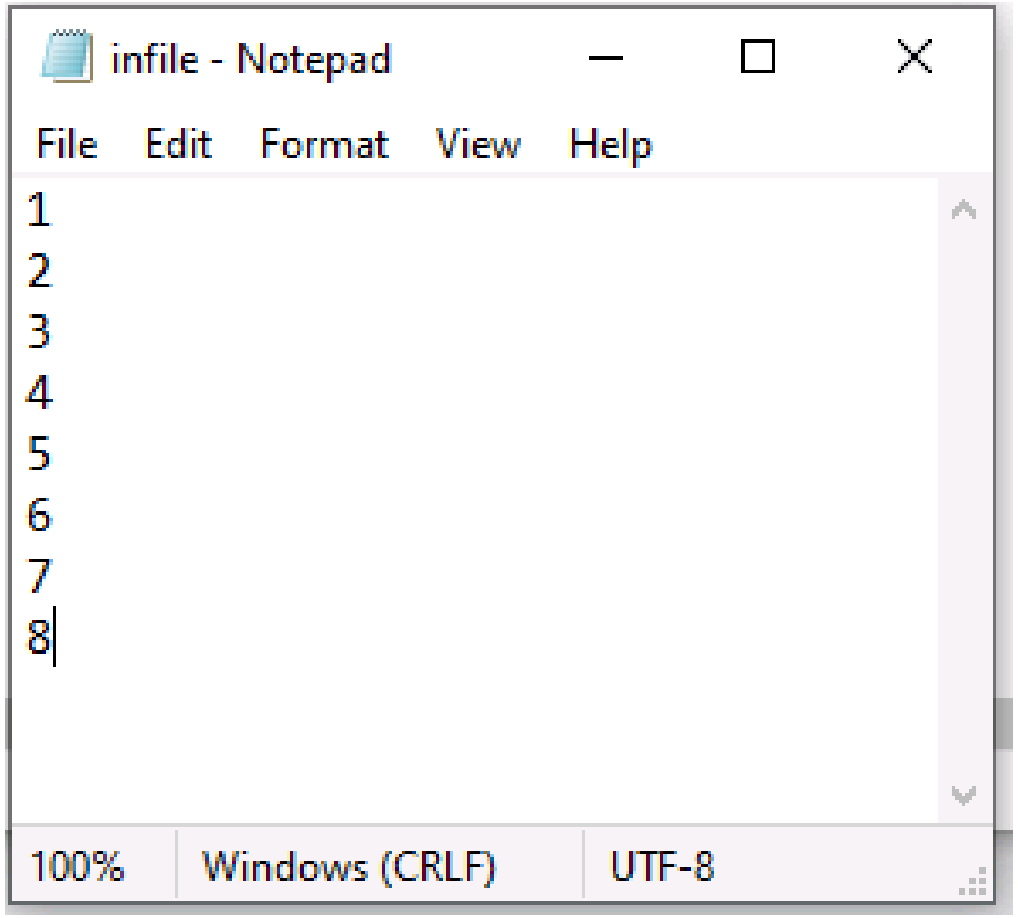
Output is performed first, (then you cannot perform input with this object afterwards~it wont work as required!)

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    fstream f1, f2;
    f1.open("infile.txt", ios::in | ios
::out);
    int first, second, third;
    f1 << "Finished reading data";
    f1 >> first >> second >> third;
    cout << first << second << third <<
endl;
    f1.close();
}
```

Pointer points at the start for writing the data, in truncate mode by default!



Read and write (truncate mode) on same file with different objects


infile - Notepad
File Edit Format View Help
1
2
3
4
5
6
7
8
100% Windows (CRLF) UTF-8

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    fstream f1, f2;
    f1.open("infile.txt", ios::in);
    int first, second, third;
    f1 >> first >> second >> third;
    cout << first << second << third <<
endl;
    f1.close();


    f2.open("infile.txt", ios::out);
    f2 << "Write a text" << endl;
    f2.close();
}
```

Write to console

Microsoft Visual Studio Debug Console

```
123
```

Read first three bytes

File pointer

infile - Notepad

File Edit Format View Help

```
1
2
3
4
5
6
7
8
```

100% Windows (CRLF) UTF-8

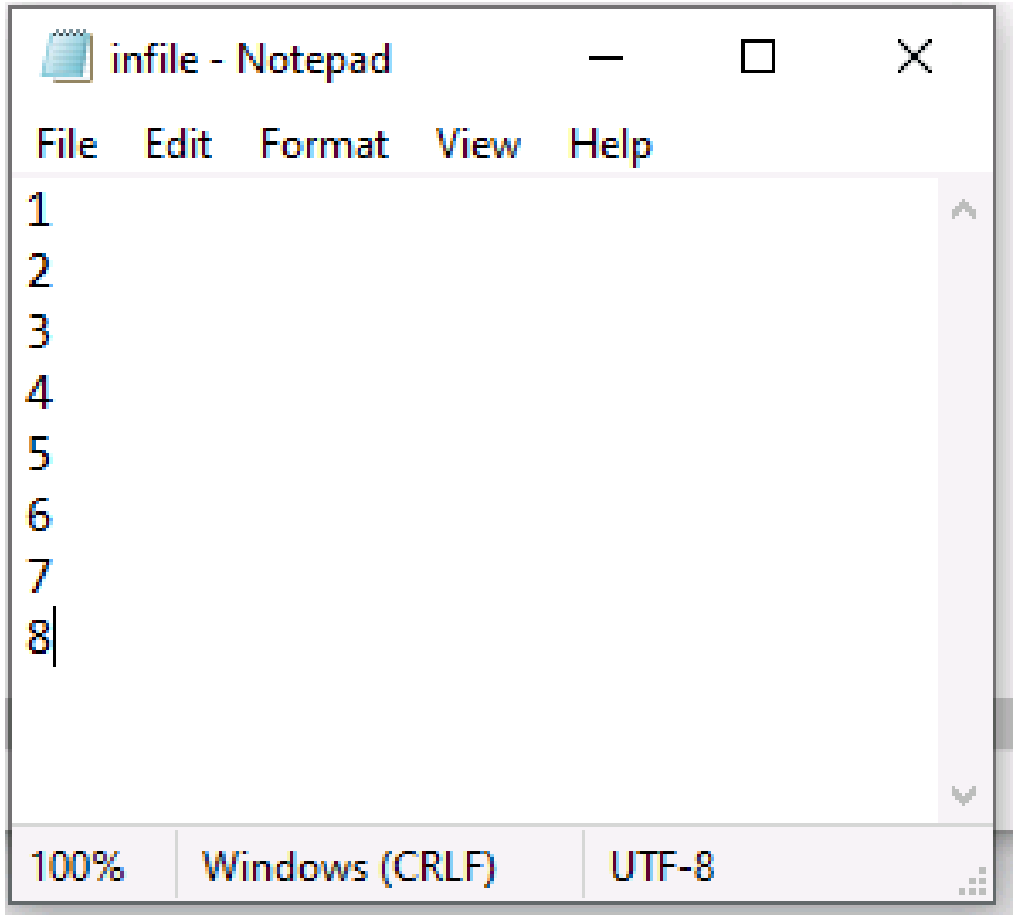
infile - Notepad

File Edit Format View Help

```
Write a text
```

Open the same file in truncate mode, reset the pointer, and write text

Read and write (append mode) on same file with different objects


infile - Notepad
File Edit Format View Help
1
2
3
4
5
6
7
8
100%
Windows (CRLF)
UTF-8

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    fstream f1, f2;
    f1.open("infile.txt", ios::in);
    int first, second, third;
    f1 >> first >> second >> third;
    cout << first << second << third <<
endl;
    f1.close();


    f2.open("infile.txt", ios::app);
    f2 << "Write a text" << endl;
    f2.close();
}
```

Write to console

Microsoft Visual Studio Debug Console

```
123
```

Read first three bytes

File pointer

infile - Notepad

File Edit Format View Help

```
1
2
3
4
5
6
7
8
```

EOF

UTF-8

infile - Notepad

File Edit Format View

```
1
2
3
4
5
6
7
8Write a text
```

Open the same file in append mode, point the pointer to EOF and write text

Some useful functions related files

Check if file exists (before reading)

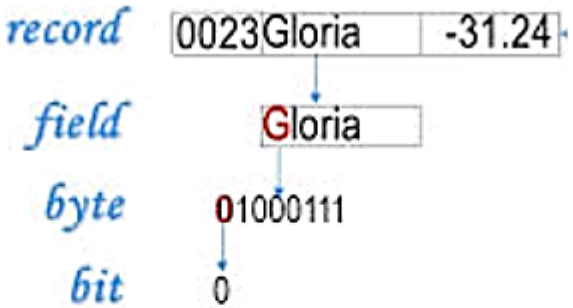
```
#include<iostream>
#include<fstream>
using namespace std;
int main(void)
{
    int data;
    ifstream fin;     // declare stream variable name
    fin.open("aa.txt", ios::in);    // open file
    if (fin.fail())
    {
      //File does not exist code here
    }
    else
    {

     }
fin.close();        //close file
return 0;
}
```

Fail() returns true if no file exists!
False, otherwise

EOF character

- C++ provides a special function, **eof( )**, that returns nonzero (meaning TRUE) when there are no more data to be read from an input file stream, and zero (meaning FALSE) otherwise.
- eof() tests whether the "end of file" flag is set on the C++ stream object.
- Before you attempt a read on an empty file the flag is not set.
- You have to perform an operation that will try to read something before the flag will be set
- For example, consider the following example in which we attempt to read from an empty file file.txt. But, as the eof flag is not set therefore the control moves inside while loop!

```
int data=0;
ifstream fin;
fin.open("file1.txt", ios::in);
while (!fin.eof())
{
        fin >> data;
}
```



✓C++ views each file as a sequence of bytes terminated by EOF-marker.

```
int data=0;
ifstream fin;

fin.open("file1.txt", ios::in);     // open file

if (fin.fail()) {}
else
{
    fin >> data;
    while (!fin.eof())       //if not at end of
    file, continue reading numbers
    {
        cout << "entered " << endl;
        cout << data << endl;    //print numbers
        to screen
        fin >> data;                //get next
        number from file
        cout << "read " << data << " from file"
        << endl;
    }
}
fin.close();        //close file
```

Fail() check for existence of file

eof() checks if the file is empty

file1 - Notepad

File Edit Format View Help

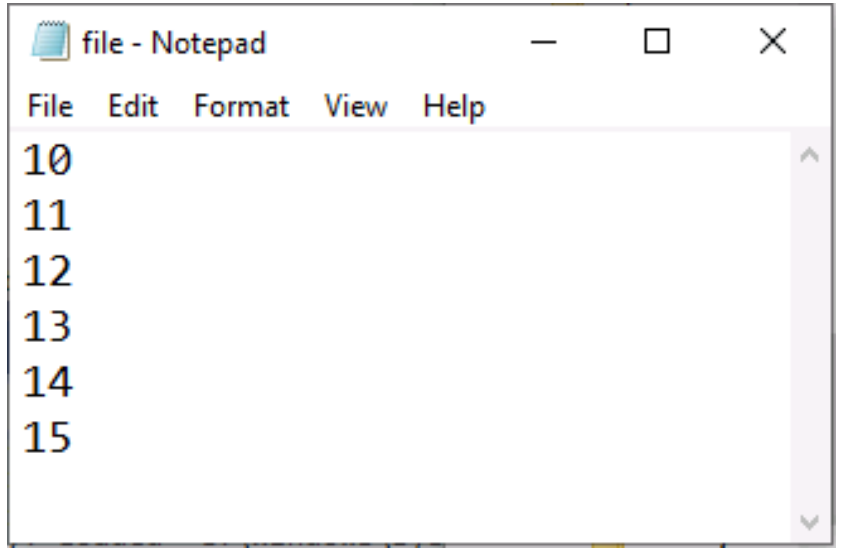
```
1
2
3
4
5
```

Microsoft Visual Studio Debug Console

```
entered
1
read 2 from file
entered
2
read 3 from file
entered
3
read 4 from file
entered
4
read 5 from file
```

Write to a file based on some condition!

```
#include<iostream>
#include<fstream>
using namespace std;
int main(void)
{
    int data=10;
    fstream f;
    f.open("file.txt", ios::out);    // open
    file
    while(data!=16)
    {
        f << data<<endl;
        data++;
    }
    f.close();        //close file
    return 0;
}
```



Some content for extra practice

Binary file

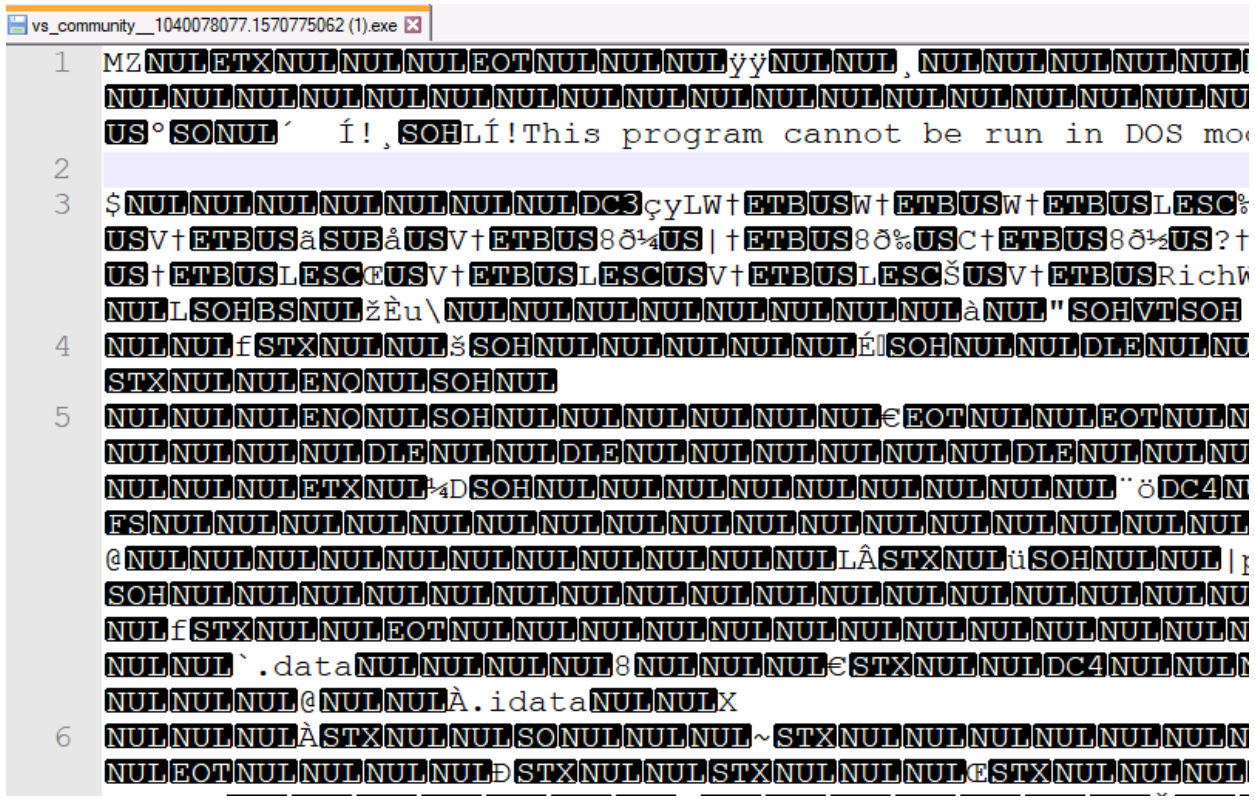
- A binary file is a type of computer file that is used to store binary data.
- It is used directly by the computer and generally can't be read by a human.
- Suppose we wanted to store the number 4,000,000,000 (4 billion). As humans, we would write it as 4000000000, or 10 ASCII characters (10 bytes). How would a computer do it?
- A single byte has 8 bits, or 2^8 (256) possible values. 4 bytes gives us 2^32 bits, or roughly 4 billion values. So, we could store the number 4 billion in only 4 bytes.
- As you can see, storing numeric data in the computer's format saves space. It also saves computational effort — the computer does not have to convert a number between binary and ASCII.
- You can write a few numbers to disk using formatted I/O, but if you're storing a large amount of numerical data it's more efficient to use binary I/O, in which numbers are stored as they are in the computer's RAM memory, rather than as strings of characters.

Summary

- **Why binary files?**
- Input and output are much faster using binary data. Converting a 32-bit integer to characters takes time. If a file (such as an image file) contains millions of numbers the accumulated conversion time is significant. Computer games would slow to a crawl if their data were stored in character form.
- A binary file is usually very much smaller than a text file that contains an equivalent amount of data. For image, video, and audio data this is important. Small files save storage space, can be transmitted faster, and are processed faster. I/O with smaller files is faster, too, since there are fewer bytes to move.
- **Example of binary files:** *.exe*

What if we open an exe file in notepad or any text editor?

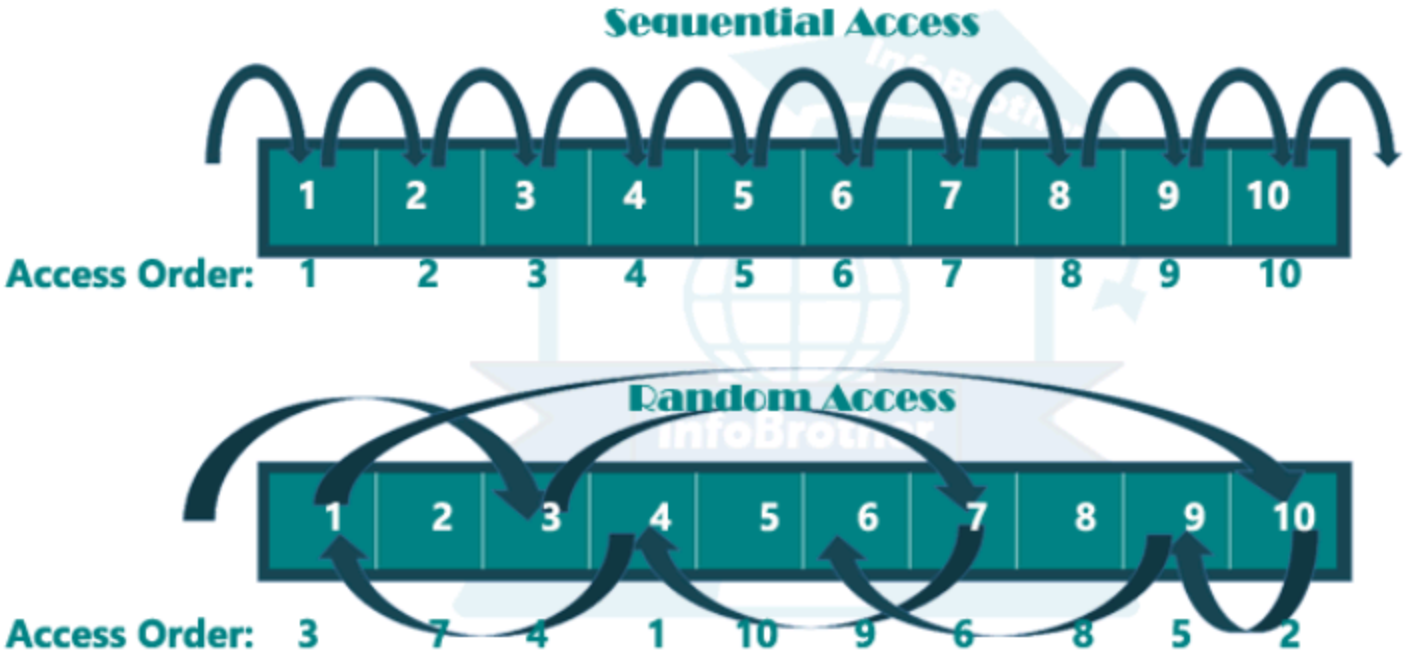
- Exe is not text.
- It includes characters quite a lot of characters within non human-readable range
- Say 0x0000_0000 is a code that your processor understands as an ADD instruction (reality is little complex than this, but very similar). Assume ADD is the first instruction in your exe. Now first 4 bytes of your exe file will have this data 0x0000_0000.



Binary vs text files

- All files are binary; the data within them is a binary representation of some information.
- If you have to store a large amount of text then the file will contain the binary representation of that text.
- The difference between a "binary file" and a "text file" is that creating the latter involves converting data to a text form before saving it. This is typically done so humans can read it.

Sequential vs random access

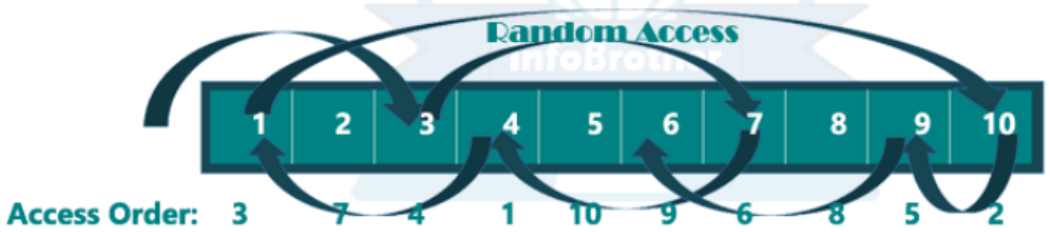


File pointers
(separate pointers for read and write)

Streams and pointers associated with them

- For filestreams, there is a joint pointer that works for both input/output (moving one will automatically move the other one)
- For stringstream, there is a separate pointer for read and write operation
- Lets go through each..

Random access



- Random access treats file as a big array.
- We can read from anywhere and write to anywhere in the file
- File pointer is involved to allow random access.
- Each file has two pointers: **input** and **output pointer**
- ***A joint file position is maintained for both the input sequence and the output sequence.***
- ***So, seekg and seekp are interchangeable for file streams. However, this is not true for other types of streams (e.g stringstream), as they may hold separate pointers for the put and get positions.***
- **seekg()** and **tellg()** functions are for input streams (ifstream) and **seekp()** and **tellp()** functions are for output streams (ofstream).
- The tellg() function returns the current position of the get pointer

Special function to move File-Pointer within the File:

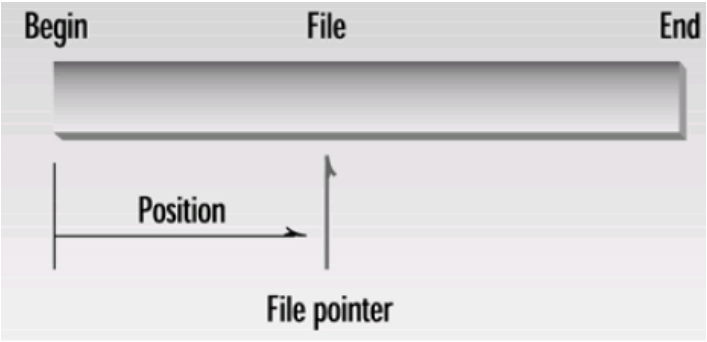
Function	Syntax	Explanation
seekg()	`fileObject.seekg(long_num, origin);`	We can move input pointer to a specified location for reading by using this function. fileObject is the pointer to the file that we want to access. long_num is the number of bytes in the file we want to skip. and origin is a value that tells to compiler, where to begin the skipping of bytes.
seekp()	`fileObject.seekp(long_num, origin);`	We can move Output pointer to a specified location by using this function. it's same like seekg() Function but in Case of writing:
tellg()	`fileObject.tellg();`	This function return the current position of Input pointer. this function don't need any argument.
tellp()	`fileObject.tellp();`	This function return the current position of Output pointer. this function don't need any argument.

File streams and random access

Seekg function's form 1: seekg(long);

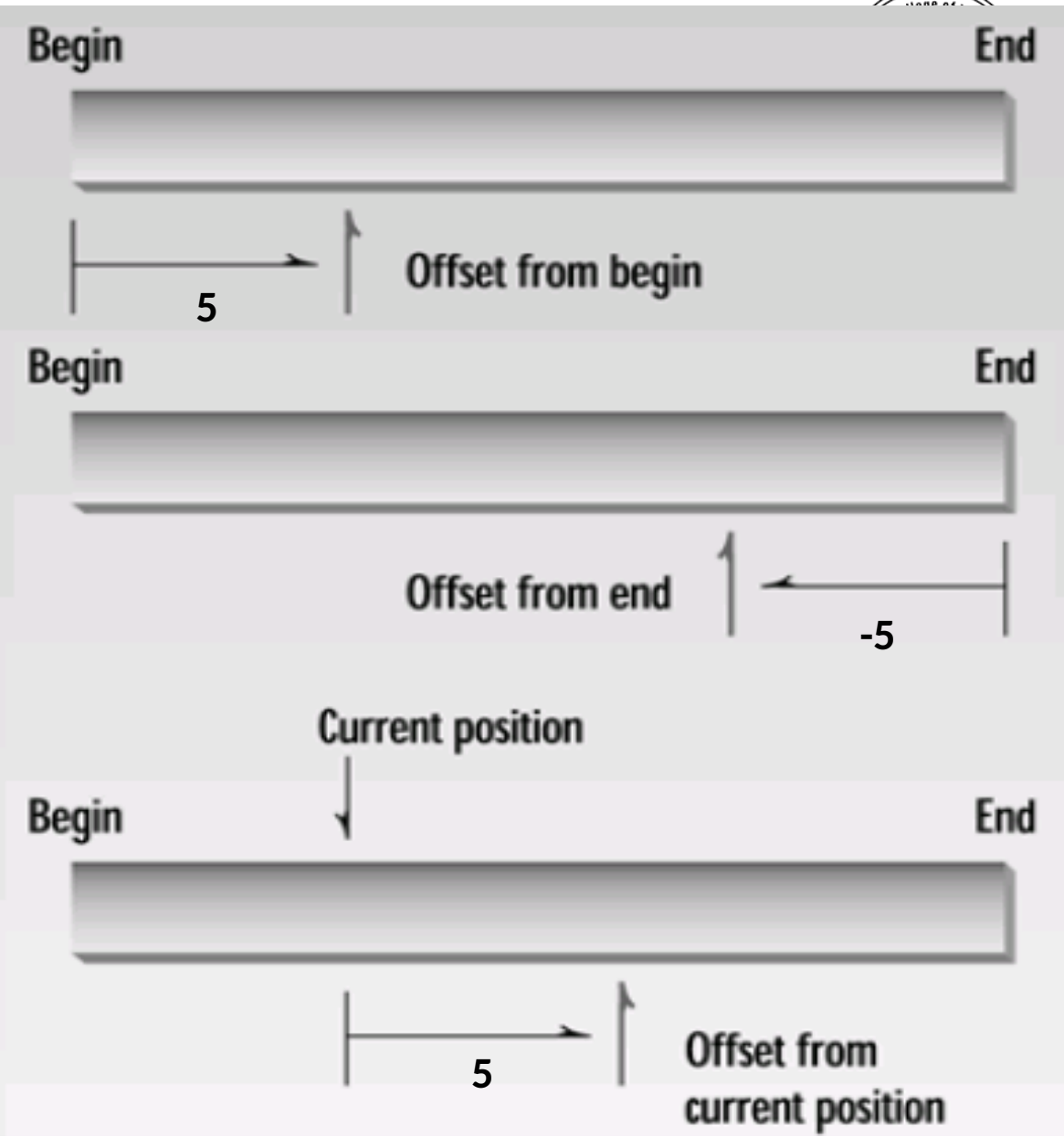
- it moves the get_pointer or put_pointer to an absolute position.

```
ifstream fin;
ofstream fout;
                    // file opening routine
fin.seekg(30);          // will move the get_pointer (in ifstream) to byte number 30 in the file
fout.seekp(30);         // will move the put_pointer (in ofstream) to byte number 30 in the file
```



Seekg form 2: seekg(offset, orig

- **Offset** is the number of bytes that you want to skip
- **Origin** is the value from where we want to start the skipping. There are three possibilities for it:
  - **Ios::beg** is the beginning of the file,
  - **Ios::cur** is the current pointer position, and
  - **Ios::end** is the end of the file.



IOS::BEG

```
fstream file;
file.open("file1.txt", ios::out);
file << "BSEF19M001";
cout << "Current position is: " << file.tellp() << endl;
```

Microsoft Visual Studio Debug Console

```
Current position is: 10
```

0	1	2	3	4	5	6	7	8	9	10
B	S	E	F	1	9	A	0	0	1	EOF
0501	0502	0503	0504	0505	0506	..				

File pointer

file1 - Notepad

File Edit Format View Help

BSEF19A001

```
file.seekp(6, ios::beg);
file << "A";
file.close();
```

IOS::CUR

```
fstream file;
file.open("file1.txt", ios::out);
file << "BSEF19M002";
cout << "Current position is: " << file.tellp() << endl;
```

Microsoft Visual Studio Debug Console

```
Current position is: 10
```

0	1	2	3	4	5	6	7	8	9	10
B	S	E	F	1	9	M	0	0	2	EOF
0501	0502	0503	0504	0505	0506	..				

File pointer

```
file.seekp(3, ios::cur);
file << "BSEF19A002";
file.close();
```

file1 - Notepad

File Edit Format View Help

BSEF19M002 BSEF19A002

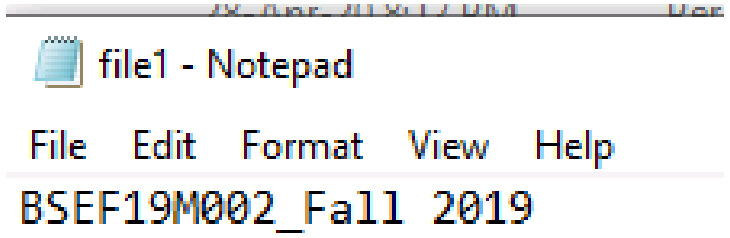
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
B	S	E	F	1	9	M	0	0	2				B	S	E	F	1	9	A	0	0	2	EOF
01	02	03	04	05	06	..	..																

Current position

Skipping 3 bytes starting from the current position

IOS::END *(offset must be a negative value! Because from end we always have to move backward!)*

```
fstream file;
file.open("file1.txt", ios::out);
file << "BSEF19M002_*";
file.seekg(-1, ios::end);
file << "Fall 2019";
file.close();
```



Start printing text from here

Because of -1 move backward one step!

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
B	S	E	F	1	9	M	0	0	2	_	*	EOF											
01	02	03	04	05	06	..	..																

end position

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
B	S	E	F	1	9	M	0	0	2	_	F	a	l	l		2	0	1	9	EOF			
01	02	03	04	05	06	..	..																

Why stringstreams are useful?

- As we discussed that filestream treats both get and put pointer jointly, therefore we will not be able to see the effect of get and put pointer separately
- E.g. when we will move get pointer, then put pointer will also get moved automatically!
- Therefore, to realize the advantages of get and put pointers, we will see how to use them with string stream!
- String stream is a stream (method for dealing with files) that is extremely useful for **parsing input file** and **using get/put pointer for random access**

String stream

- Stream is a sequence of bytes.
- A stringstream associates a string object with a stream allowing you to read from the string as if it were a stream (like cin)
- We have to include <sstream> library

String stream: example

```
std::stringstream s("0123456789");
std::cout << "put pointer: " << s.tellp() << std::endl;
std::cout << "get pointer: " << s.tellg() << std::endl;
std::cout << std::endl;
s.seekp(2);
string t,t1;

std::cout << "put pointer: " << s.tellp() << std::endl;
std::cout << "get pointer: " << s.tellg() << std::endl;
std::cout << std::endl;
s.seekg(4);
s >> t;
cout <<"t "<< t << endl;
cout << "s " << s.str() << endl;
s.clear();
std::cout << "put pointer: " << s.tellp() << std::endl;
std::cout << "get pointer: " << s.tellg() << std::endl;
std::cout << std::endl;
s.seekp(0, ios::end);
s << " number";
cout << "s " << s.str() << endl;
std::cout << "put pointer: " << s.tellp() << std::endl;
std::cout << "get pointer: " << s.tellg() << std::endl;
std::cout << std::endl;
```

Microsoft Visual Studio Debug Console

```
put pointer: 0
get pointer: 0

put pointer: 2
get pointer: 0

t 456789
s 0123456789
put pointer: 2
get pointer: 10

s 0123456789 number
put pointer: 17
get pointer: 10
```

String stream: example

```
std::stringstream s("0123456789");
std::cout << "put pointer: " << s.tellp() << std::endl;
std::cout << "get pointer: " << s.tellg() << std::endl;
std::cout << std::endl;
```

Microsoft Visual Studio Debug Console

```
put pointer: 0
get pointer: 0
```

String stream: example

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	EOF
0501	0502	0503	0504	0505	0506	..				

get *put*

```
s.seekp(2);
string t,t1;

std::cout << "put pointer: " << s.tellp() << std::endl;
std::cout << "get pointer: " << s.tellg() << std::endl;
std::cout << std::endl;
```

```
put pointer: 2
get pointer: 0
```

String stream: example

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	EOF
0501	0502	0503	0504	0505	0506	..				

put *get*

```
s.seekg(4);
s >> t;
cout <<"t "<< t << endl;
cout << "s " << s.str() << endl;
```

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	EOF
0501	0502	0503	0504	0505	0506	..				

put *get*

```
t 456789
s 0123456789
```

String stream: example

```
s.clear();
std::cout << "put pointer: " << s.tellp() << std::endl;
std::cout << "get pointer: " << s.tellg() << std::endl;
std::cout << std::endl;
```

```
put pointer: 2
get pointer: 10
```

String stream: example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	1	2	3	4	5	6	7	8	9		n	u	m	b	e	r	EOF	
0501	0502	0503	0504	0505	0506	..												

get *put*

```
s.seekp(0, ios::end);
s << " number";
cout << "s " << s.str() << endl;
std::cout << "put pointer: " << s.tellp() << std::endl;
std::cout << "get pointer: " << s.tellg() << std::endl;
std::cout << std::endl;
```

```
s 0123456789 number
put pointer: 17
get pointer: 10
```

Usefulness

- Used to easily parse input data
- Can be used to parse input while file read/write operations

```
stringstream ss;
ss << "2.0 35 a";
double x, int y; char z;
ss >> x >> y >> z;
```

x	2.0
y	35
z	'a'

Convert stringstream to string

- Use the .str() member function to create a large string from many value (i.e. return a string with the contents of whatever is in the stream)

```
#include using namespace std;
int main()
{
  stringstream ss;
  ss << 2.0 << " " << 35;
  ss << " " << 'a';
  string s = ss.str();
  return 0;
}
```

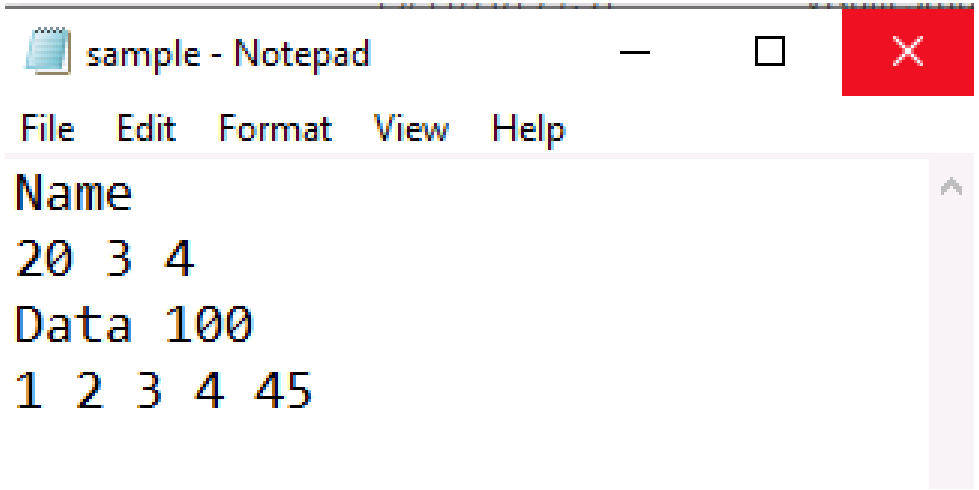
s "2.0 35 a"

Parse input: convert from string to other data type

```
string text;
int num;
double val;
stringstream ss("Hello 103 2.0");
ss >> text >> num >> val;
```

Parse input: when there are varying number of elements on each line

```
fstream myfile; int total_words = 0; int
num_lines = 0;
myfile.open("sample.txt", ios::in);

string str;
while (getline(myfile, str))
{
        stringstream ss(str);
        string word;
        while (ss >> word)
        {
                total_words++;
        }
                num_lines++;
}
cout << total_words << endl;
cout << num_lines << endl;
myfile.close();
```



Parse input: extract words that are delimited by comma

```
fstream myfile; int total_words = 0; int
num_lines = 0;
myfile.open("sample.txt", ios::in);

string input = "abc,def,ghi";
stringstream ss(input);
string token;

while (getline(ss, token, ','))
{
        cout << token << endl;
}
myfile.close();
```

Microsoft Visual Studio Debug Console

```
abc
def
ghi
```

End

Programming fundamentals

Lecture 25: Structures (structs)

Recap

- Random access in files
- Seekg and seekp

Agenda

- Structures

Introduction

- Sometimes it is useful to have a collection of values of different types and to n as a single item.
- Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.
- **For example:** You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables

  name, citNo, salary

- to store these information separately.
- However, in the future, you would want to store information about multiple persons.
- Now, you'd need to create different variables for each information per person:

  name1, citNo1, salary1, name2, citNo2, salary2.....

- You can easily visualize how big and messy the code would look

- A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

- This collection of all related information under a single name Person is a structure.

How to define struct?

The ***struct keyword*** defines a structure type followed by an identifier (name of the structure).

Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example:

```
struct Person
{
   char name[50];
   int age;
   float salary;
};
```

name[50]	age	salary
Person		

Here a structure person is defined which has three members: name, age and salary.

- Structure definition is only the blueprint for the creating of variables. You can imagine it as a datatype. When you define an integer as below:

```
int foo;
```

- The int specifies that, variable foo can hold integer element only.
- Similarly, structure definition only specifies that, what property a structure variable holds when it is defined

int foo

Person bill

- ***When a structure is created, no memory is allocated.***

int foo

Person bill

```
struct Person
{
    char name[50];
    int age;
    float salary;
};
```

memory of float is 4 bytes

memory of int is 4 bytes

memory of char is 1 byte. 50 cells array has 50 bytes

58 bytes of memory is allocated for structure variable `bil`

How to access members of a structure?

The members of structure variable is accessed using a dot (.) operator.

Suppose, you want to access age of structure variable bill and assign it 50 to it. You can perform this task by using following code below:

```
bill.age = 50;
```

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

struct Person
{
    char name[50];
    int age;
    float salary;
};
```

```cpp
int main()
{
    Person p1;

    cout << "Enter name: ";
    cin >> p1.name;
    cout << "Enter age: ";
    cin >> p1.age;
    cout << "Enter salary: ";
    cin >> p1.salary;

    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p1.name << endl;
    cout << "Age: " << p1.age << endl;
    cout << "Salary: " << p1.salary;

    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Enter name: Amna
Enter age: 35
Enter salary: 50000

Displaying Information.
Name: Amna
Age: 35
Salary: 50000
```

Array of structs

We can also make an array of structures. In the first example in structures, we stored the data of 3 persons. Now suppose we need to store the data of 100 such persons. Declaring 100 separate variables of the structure is definitely not a good option. For that, we need to create an array of structures.

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

struct Person
{
    char name[50];
    int age;
    float salary;
};
```

0	1	2	3	4
p1[0]	p1[1]	p1[2]	p1[3]	p1[4]
0501	0502	0503	0504	0505

Contents of an array would be a struct

```
int main()
{
    Person p1[5];
    int i;
    for (i = 0; i < 5; i++) {
        cout << "Person" << i + 1 << endl;
        cout << "Enter name" << endl;
        cin >> p1[i].name;
        cout << "Enter age" << endl;
        cin >> p1[i].age;
        cout << "Enter salary" << endl;
        cin >> p1[i].salary;
    }
    for (i = 0; i < 5; i++) {
        cout << "Person" << i + 1 << endl;
        cout << "Details" << endl;
        cout << p1[i].name;
        cout << p1[i].age;
        cout << p1[i].salary;
    }
    return 0;
}
```

Structs and functions

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

struct Person
{
    char name[50];
    int age;
    float salary;
};
void display(Person p)
{
    cout << "\nDisplaying
Information." << endl;
    cout << "Name: " << p.name << endl
;
    cout << "Age: " << p.age << endl;
    cout << "Salary: " << p.salary;

}
```

```
int main()
{
    Person p1;

    cout << "Enter name: ";
    cin >> p1.name;
    cout << "Enter age: ";
    cin >> p1.age;
    cout << "Enter salary: ";
    cin >> p1.salary;
    display(p1);
    return 0;
}
```

Programming fundamentals

We studied that size of struct is computed by adding up all primitive data types used

Person bill

```
struct Person
{
    char c;
    int age;
};
```

memory of int is 4 bytes

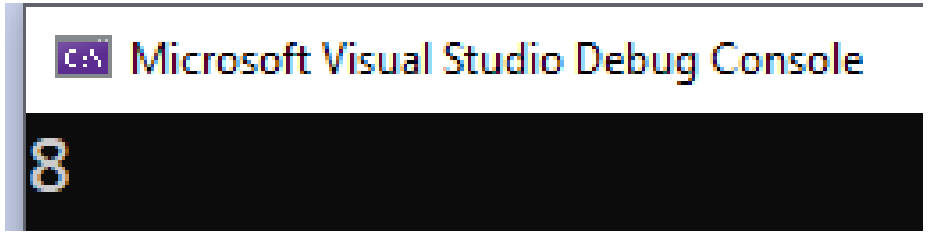
memory of char is 1 byte

5 bytes of memory is allocated for structure variable bill

But if you run code and find size using sizeof function then the answer will be slightly different

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

struct Person
{
    char c;
    int age;
};

int main()
{
    Person p;
    cout << sizeof(p);
return 0;
}
```

It should be 5! But the reason that why it is 8 is that compiler uses extra bytes (8-5) in doing memory management (padding)



What is padding?

- Padding means to consider chunks of 4 bytes (in case of 32 but system) everytime while accessing anything
- That means if we came across char variable and we know it consumes 1 byte..but our system will access it in 1 word (4 bytes)
- Therefore, the compiler will use 1 byte in storing char and 3 bytes taken as extra bytes (padded at the end to make the chunk of 4 byte)

```
struct Person
{
    char c;
    int age;
};
```

8 bytes of memory is allocated for structure variable bill

8 bytes		
c	Padded bytes	age
1 byte	3 bytes	4 bytes

Note

- If the struct has only one variable of one primitive type..then there is no need for padding..
- Padding usually applies when there are multiple variables
- In summary, extra bytes are consumed in organizing the memory (padding)