|  |  |  |
| --- | --- | --- |
| Colour | Meaning | Example |
| COLOUR | Words With Special Meaning | Predefined Function |
| **COLOUR** | Begins Preprocessor Directive | #include |
| COLOUR | Header File | <iostream> |
| COLOUR | Key Words / Special Character or Symbol Used With the Key Words | setw(x) |
| COLOUR | Things Should Be Filled | cin.ignore(‘Number of Characters Need to Skip’, ‘UNTIL’); |
| **COLOUR** | Actual Variable / Actual Data / Actual Number / Actual String | int length = 12; // Length in Inches |

Contents

[1. Beginning 1 (Preprocessor Directive & Predefined Function / Library Function) 2](#_Toc187263908)

[2. Beginning 2 (User-Defined Function) 3](#_Toc187263909)

[3. Basic Programming 7](#_Toc187263910)

[4. Key Words (Data Type) 10](#_Toc187263911)

[5. Key Words (Input & Output) 13](#_Toc187263912)

[6. Key Words (Selection & Loop) 16](#_Toc187263913)

[7. Key Words (Math Function) 19](#_Toc187263914)

[8. Key Words (Time Function) 20](#_Toc187263915)

[9. Key Words (Data Conversion Function) 21](#_Toc187263916)

[10. Key Words (Character Classification and Conversion Function) 22](#_Toc187263917)

[11. Array 23](#_Toc187263918)

[12. Key Words (File Operation) 26](#_Toc187263919)

[13. Pointers 29](#_Toc187263920)

[14. Structured Data 33](#_Toc187263921)

# Beginning 1 (Preprocessor Directive & Predefined Function / Library Function)

|  |  |  |
| --- | --- | --- |
| **Header File** | * **Header File** provides the function (Standard Function / Predefined Function / Library Function) we need. * Library Functions are the “built-in" functions that come with the compiler. * Different types of **Header File** store different types of Predefined Function can be used. | |
| #include <iostream> | * <iostream> is an example of **Header File** which can provide basic input and output services for C++ programs. * Functions such as cout and cin can be used if we put the <iostream> as one of our Header File. |
| #include <string> | * Purpose : - * Function string can be used. |
| #include <cstring> | * Purpose : Numerous functions for handling C-strings. * Function strlen(), strcat(), strcpy(), strstr() and strcmp() can be used. |
| #include <iomanip> | * Purpose : - * Function setw() and setprecision() can be used. |
| #include <cmath> | * Purpose : Math functions. * Function abs(x), fabs(x), ceil(x), floor(x), pow(x, y), pow10(x), sqrt(x), exp(x), log(x), log10(x), sin(x), cos(x) and tan(x) can be used. |
| #include <ctime> | * Purpose : Time functions. * Function time() can be used. |
| #include <cstdlib> | * Purpose : Data conversion. * Function rand(), srand(), atoi(), atoll(), atof() and exit() can be used. |
| #include <cctype> | * Purpose : Character classification and conversion. * Function isalpha(), isalnum(), isdigit(), islower(), isprint(), ispunct(), insupper(), isspace(), toupper() and tolower() can be used. |
| #include <fstream> | * Purpose : Use when file input and output. * Function ifstream, ofstream and fstream can be used. (Data Type / Variable Declaration) |
| **Definition File**  **(Optional)** | #define ‘A’ ‘B’  ‘A’ (Name)  ‘B’ (Meaning / Value) | * **Definition File** gives a name to a constant value. |
| * Example :   #define ‘Word1’ ‘Word2’   * ‘Word1’ has been defined as the ‘Word2’. * Example :   #define PI 3.14159   * The word PI has been defined as 3.14159 at the beginning. * Thus, we have no need to put the value 3.14159 every time in our program if needed. We can just put PI to represent the value 3.14159. |
| **Other** | using namespace std; |  |
| **Main Structure (Basic / Common) :**  #include <iostream>  using namespace std;  int main ()  {  return 0;  } | | |

# Beginning 2 (User-Defined Function)

A diagram of a function

Description automatically generated A diagram of a function

Description automatically generated

* **User-Defined Functions** are created by you, the programmer.
* **Function Call :** Statement causes a function to execute.
* **Function Definition :** Statement that makes up a function.
  + **Return Type :** Data type of the value that function returns to the part of the program that calls it.
  + **Name :** Name of the function. Function names follow the same rules as variables.
  + **Parameter list :** Variables containing values passed to the function.
  + **Body :** Statements that perform the function’s task, enclosed in {}.

1. **Function Definition :**
2. **Return Type :** Data type of the value that function returns to the part of the program that calls it.

|  |  |  |
| --- | --- | --- |
| **Return Type** | **Explain** | **Example** |
| int  double  … | If a function returns a value, the type of the value must be indicated. |  |
| void | If a function does not return a value, its return type is void. |  |
| **Key Words** | **Explain** | **Example** |
| return | * Used to return a value to the Function at the last statement. * return ‘Variable / Value / Expression / Operation’; * The Function will equal to the ‘Variable / Value’ you return. |  |

1. **Function Name :** Name of the function. Function names follow the same rules as variables.

* If want to call a function, use the function name followed by () and ;
* ‘FunctionName’ (‘ParameterList (If Have)’);
* Example :

A screenshot of a computer program

Description automatically generated  

1. **Parameter List :** Variables containing values passed to the function.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type Parameter (Argument)** | **Explain** | **Example** | **Example** |
| Formal Parameter  (Formal Argument) | * Use when build a function. |  |  |
| Actual Parameter  (Actual Argument) | * Use when call a function. * Actual Parameter can be an actual value also. * The number of arguments in the call must match the prototype and definition. |  |

1. **Function Body :** Statements that perform the function’s task, enclosed in {}.

* We can declare a variable in a User-Defined Function also.

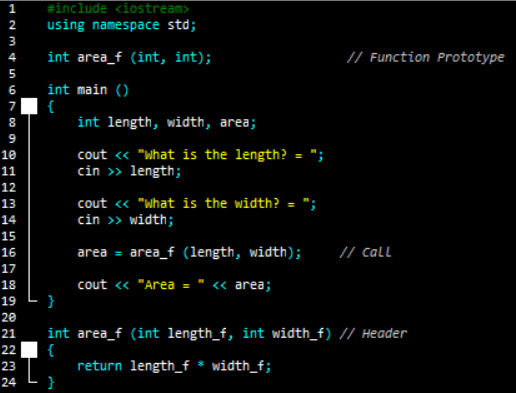
A computer screen shot of numbers

Description automatically generated

1. **Function Prototype :** Ways to notify the compiler about a function before a call to the function.

A screenshot of a computer program

Description automatically generated A screenshot of a computer code

Description automatically generated 

**Header**

**Call**

**Function prototypes**

**Header**

**Header**

* Place prototypes near the top of program.
* Program must include either prototype or full function definition before any call to the function. If not, it will compile error.
* When using prototypes, can place function definitions in any order in the source file.
* The prototype must include the data type of each parameter inside its parentheses.

The header must include a declaration for each parameter in its ().

A close up of a text

Description automatically generated

1. **Local and Global Variables :**

* Local variables are not automatically initialized. They must be initialized by a programmer.
* Global variables (not constants) are automatically initialized to 0 (numeric) or NULL (character) when the variable is defined.

|  |  |  |
| --- | --- | --- |
| **Variables** | **Explain** | **Example** |
| Local Variable | * Variables defined inside a function. * They are hidden from the statements in other functions, which normally cannot access them.   Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.   * A function’s local variables exist only while the function is executing. This is known as the lifetime of a local variable.   When the function begins, its local variables and its parameter variables are created in memory and when the function ends, the local variables and parameter variables are destroyed. |  |
| * **Other :** Static Local Variable. * Static Local Variables retain the contents of variables between function calls. * Local Variable in a function will not destroy after a call. * static ‘DataType’ ‘Variable’; |  |
| Global Variable | Variable defined outside all the functions in a program. |  |

1. **Reference Variable :**

|  |  |  |
| --- | --- | --- |
| **Explain** | **Example** | **Example** |
| * Defined with an Ampersand (&). * ‘DataType’ &’Variable1’ = ‘Variable3’, &’Variable2’ = ‘Variable3’; * ‘Variable1’ and ‘Variable2’ become the reference variable to ‘Variable3’. * All of the changes apply on the ‘Variable1’, ‘Variable2’ or ‘Variable3’ will bring effect to another two variables also. | A screenshot of a computer program  Description automatically generated | A computer screen shot of a program code  Description automatically generated |

1. **Default Argument :** An argument that is passed automatically to a parameter if the argument is missing on the function call.

|  |  |  |
| --- | --- | --- |
| **Explain** | | **Example** |
| * Must be a constant declared in prototype. | A black and white text  Description automatically generated |  |
| * Can be declared in header if no prototype. |  |

1. **Overload Functions :**

* Have the same name but different parameter lists.
* It can be used to create functions that perform the same task but take different parameter types or different number of parameters.
* The compiler will determine which version of function to call by argument and parameter lists.

A white text with black text

Description automatically generated

# Basic Programming

1. **Punctuation :**

|  |  |
| --- | --- |
| , | * Separate items in a list. |
| ; | * Mark the end of a statement. / Ends a programming statement. |

1. **Operators (Used to perform operations on data.) :**

* 1st Order → ()
* 2nd Order → Unary Operator : a++ or a--
* 3rd Order → Arithmetic Operators (Multiply / Divide / Modulus ) : \* / %
* 4th Order → Arithmetic Operators (Plus / Minus) : + -
* Solve from left to right if the order to solve is same.

|  |  |  |
| --- | --- | --- |
| **Arithmetic Operator** | + | * Plus. |
| - | * Minus. |
| \* | * Multiply. |
| / | * Divide. |
| % | * Modulus. (Calculate the remainder after doing the division.) * Example :   7 % 3 = 1  7 % 5 = 2  1 % 2 = 1  2 % 2 = 0  2 % 2.0 = Error (Must use integer when doing the calculation.) |
| **Assignment Operator** | = | * CAUTION : IT IS DIFFERENCE WITH == |
| **Unary Operator (Unary Negation)** | Postfix Expression | * a++ / a-- (Increment / Decrement) * Do the operation first by using value ‘a’ original before plus / minus the value of a with 1. |
| Prefix Expression | * ++a / --a * Plus / minus the value of a with 1 first before do the operation by using value ‘a’ after plus / minus with 1. |
| **Combined Assignment Operator** | += | * a = a + 3 same as a += 3 * a = a + (3 / b) same as a += 3 / b |
| -= | * a = a - 3 same as a -= 3 |
| \*= | * a = a \* 3 same as a \*= 3 |
| /= | * a = a / 3 same as a /= 3 |
| %= | * a = a % 3 same as a %= 3 |
| **Logical Operator** | && | * And. |
| || | * Or. |
| ! | * Not. * !‘Non-Zero Integer’ = 0 |
| **Relational Operator** | > | * More Than. |
| < | * Less Than. |
| >= | * More and Equal Than. |
| <= | * Less and Equal Than. |
| == | * Equal Than. |
| != | * Not Equal Than. |

1. **Type Conversion :**

* Hierarchy of Types / Conversion Rules :

**A black text on a white background

Description automatically generated** A screenshot of a computer code

Description automatically generated

1. **Special Character :**

|  |  |  |
| --- | --- | --- |
| **Double Slash** | // | * Begins a comment. |
| **Pound Sign** | # | * Begins preprocessor directive. * Example : #include #define #if #then #else #line |
| **Open Brackets**  **Close Brackets** | < > | * Encloses filename used in #include directive. * Example : #include <iostream> |
| **Open Parentheses**  **Close Parentheses** | ( ) | * Used when naming function. |
| **Open Braces**  **Close Braces** | { } | * Encloses a group of statements.   A screen shot of a computer  Description automatically generated   * A single statement that uses three lines. |
| **Open Quote Marks**  **Close Quote Marks** | “ “ | * Encloses string of characters.   A close up of a word  Description automatically generated   * Here is a string literal. |
| **Semicolon** | ; | * Ends a programming statement (line).   A math equations with green text  Description automatically generated with medium confidence   * Here is four samples line which line is blank. |

1. **Comments :**

* Are written for persons reading the source code of the program to :
* Indicate the purpose of the program.
* Describe the use of variables.
* Explain complex sections of code.
* Are ignored by the compiler.

|  |  |  |
| --- | --- | --- |
| **Single-Line Comments** | // | * Begins with // through to the end of line * Example :   int length = 12; // Length in Inches  int width = 15; // Width in Inches  int area; // Calculated Area  area = length \* width; // Calculate Rectangle Area |
| **Multi-Line Comments**  (Also, can use for Single-Line Comments) | /\* \*/ | * Begin with /\* and end with \*/ * Example :   /\*---------------------------------------  Here's a multi-line comment.  ---------------------------------------\*/   * Example :   /\* Here’s a single-line comment. \*/ |

1. **Variables / Identifiers :**

A black screen with white text

Description automatically generated

* The first character of the variables cannot be a number.
* The character of the variables cannot include symbols other than \_ and $.
* The name of variables cannot be the Key Words / Reserved Words such as int and char.

# Key Words (Data Type)

1. **Data Type’s Range :**

* 1byte = 8bit
* 1bit = 0 or 1 (Possible Value of each bit = 2)
* 1bit at the left side (Most Significant Value) will be used as a sign bit to store the sign (+ or -) if required.
* The range of positive value will be minus by 1 because the range includes value 0.
* If the sign (negative value : - ) is not required, we can put the word ‘unsigned’ in front a Data Types Key Words.
* The range of the value that can be stored will not include negative value.
* The range of positive values that can be stored will double up.
* There is no need to release 1 bit to store the sign anymore.

1. **Key Words (Data Type - Basic) :**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Range** | **Explain** | **Example** |
| 1a. | char  signed char | 1byte (8bit) | * -(2^7) to (2^7)-1 * -128 to 127 | * Mean Character. * Used for the storage of a letter. |  |
| 1b. | unsigned char | 1byte (8bit) | * 0 to (2^8)-1 * 0 to 255 | * **Other :** Use char Storing C-String (Array) * char ‘Variable’[‘Size’] = “’String’”; * ‘Size’ must be the total character of ‘String’ + 1 because each string will have a \0 or 0 at the end of the string. * Example :     ‘Size’ = 10 |
|  | | | | | |
| 2a. | short (short int)  signed short | 2byte (16bit) | * -(2^15) to (2^15)-1 * -32768 to 32767 | * Mean Integer. * Used for the storage of numbers. * If we use int to store a floating point, the system will not round automatic and it will ignore the fractional part directly. (Example : 3.88 >>> 3) | Same as int but the range smaller. |
| 2b. | unsigned short | 2byte (16bit) | * 0 to (2^16)-1 * 0 to 65535 |
| 2c. | int  signed int | 4byte (32bit) | * -(2^31) to (2^31)-1 * -2,147,483,648 to 2,147,483,647. |  |
| 2d. | unsigned int | 4byte (32bit) | * 0 to (2^32)-1 * 0 to 4,294,967,295 |
| 2e. | long (long int)  signed long | 4-8byte (32-64bit)  Depends on compiler |  | Same as int but the range larger. |
| 2f. | unsigned long | 4-8byte (32-64bit)  Depends on compiler |  |
|  | | | | | |
| 3a. | float | 4byte (32bit) |  | * Used to store decimals. * If we didn’t set the significant digit by using setprecision(), the default significant digits will be 6. * It will be rounded automatic if more than 6 significant digits. * The number of decimal places needed can be set by using fixed and setprecision(). |  |
| 3b. | double | 8byte (64bit) |  | Same as float but the range of precision larger.  (The difference between float and double is the range of precision but not decimal place.) |
| 3c. | long double | 8byte (64bit) |  |
|  | | | | | |
| 4. | bool | 1byte |  | * Represents values that are true or false. * true = 1 ; false = 0 * true = Non-Zero Value ; false = Zero Value |  |
| **No.** | **Key Words** | **Bit Width** | **Explain** | | **Example** |
| 5. | const |  | * Used declare constants with a specific type. (Similar With #define) * The content of const cannot be changed during program execution.   If trying to change the content of const in a program, the system will show error. | |  |
|  | | | | | |
| 6a. | static\_cast <> () |  | * Used for manual data type conversion.   static\_cast <’DataType wanted’> (‘Variable’) | |  |
| * C-Style cast :   (‘DataType wanted’) ‘Variable’ | |
| * Prestandard C++ cast :   ‘DataType wanted’ (‘Variable’) | |
| 6b. | sizeof(x) |  | * Used to determine the size of a data type or variable. * x is the data type we need to determine its size. | |  |

1. **Key Words (Data Type - String) :**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1. | string |  | * **Header File :** #include <string> * Use to create and use string objects. |  |
|  | | | | |
| 2a. | strlen() |  | * **Header File :** #include <cstring> * Returns length of C-string. * strlen(‘Variable’); |  |
| 2b. | strcat() |  | * **Header File :** #include <cstring> * Appends a C-string to another. * strcat(‘Variable1’, ‘Variable2’); * ‘Variable1’ will become ‘Variable1’‘Variable2’. |
| 2c. | strcpy() |  | * **Header File :** #include <cstring> * Copy a C-string to another. * strcpy(‘Variable1’, ‘Variable2’); * ‘Variable1’ will become ‘Variable2’. |
| 2d. | strstr() |  | * **Header File :** #include <cstring> * Finds the first occurrence of second C-string in first C-string. Returns a pointer to match or NULL if no match. * strstr(‘Variable1’, ‘Variable2’); |
| 2e. | strcmp() |  | * **Header File :** #include <cstring> * Compare both C-string is same or not. * strcmp(‘Variable1’, ‘Variable2’); * Output will be 0 if ‘Variable1’ equal to ‘Variable2’. * Output will be more than 0 if ‘Variable1’ is larger than ‘Variable2’. * Output will be less than 0 if ‘Variable1’ is smaller than ‘Variable2’. |

# Key Words (Input & Output)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1a. | cin |  | * **Header File :** #include <iostream> * Used to read input from keyboard. * cin >> ‘Input1’ >> ‘Input2’; * The input process will end when receive SPACE / ENTER(endl). |  |
| 1b. | cout |  | * **Header File :** #include <iostream> * Used to display output on screen. * cout << ‘Output1’ << ‘Output2’; * Start a new line use endl behind << or \n in the “ ”. |
|  | | | | |
| 2a. | getline() |  | * To read an entire line of input with array. Include the whitespace. * The input process will end when receive ENTER(endl). |  |
| * When reading string data to be stored as an object of string, use getline() with two arguments : * istream object (Example : cin). * string object. * getline(‘From’, ‘To’);   Example : getline(cin, ‘Variable’);   * ‘Variable’ or ‘To’ must be string data type. |
| * When reading string data to be stored in a char Array, use getline() with two arguments : * Name of Array to store string (C-String / char). * Size of the Array. * ‘From’.getline('Variable’, ‘Size’);   Example : cin.getline('Variable’, ‘Size’);   * getline() reads one character less than size set in ‘Size’. (Concept char Array - Refer 11 - b - 1a,1b) |
| 2b. | get() |  | * To read a single character. Include the whitespace. * ‘From’.get('Variable’);   Example : cin.get(‘Variable’);   * ‘Variable’ must be a char data type. |  |
| * When reading string data to be stored in a char Array, use get() with two arguments : * Name of Array to store string (C-String / char). * Size of the Array. * ‘From’.get('Variable’, ‘Size’);   Example : cin.get(‘Variable’, ‘Size’);   * getline() reads one character less than size set in ‘Size’. (Concept char Array - Refer 11 - b - 1a,1b) |
| 2c. | ignore() |  | * To skip over unneeded characters. * ‘From’.ignore(‘Number of Characters Need to Skip’, ‘UNTIL’);   Example : cin.ignore(‘Number of Characters Need to Skip’, ‘UNTIL’);   * ignore() can accept an input but it can’t store the input into a certain variable. * Therefore, we can put a cin, getline() or get() after the ignore() based on our requirements. * The buffer from ignore() will be receive by the cin, getline() or get() after it. |  |
| 2d. | **BUFFER PROBLEM** | | * Input buffering is when data typed by a user is stored temporarily in a buffer before the program processes it.   Example of input buffer :   * ENTER(\n) during we input data. * Unacceptable / Ignored part of our input data (Only for string literal). * Example : If we input ‘zhang san’ using cin, ‘ san’ will become buffer because cin only can accept the string before the whitespace. * The buffer will exist until the next input receives it. | Refer 5. Key Words (Input & Output) – 2d - Buffer Problem in Folder |
| * The buffer left by the cin (\n) will not affect another cin (only for \n) but it can affect getline(), get() and ignore(). * getline() will discard the ENTER(\n) which is supposed to leave as a buffer if input using getline(). * cin, getline(), get() and ignore() can accept the buffer if exists before it. * If an input receives a buffer, that input will not ask input from user again because it already receives the buffer as input. |
| * To skip over unneeded characters that are still in the keyboard buffer :   ignore() can be applied to skip over (cancel) all the buffer exists before it by receiving the buffer by itself and ignore it.  It can avoid the buffer received by the cin, getline() or get() after it.  Example : cin.ignore(100, ‘\n’); can be put before cin, getline() or get() if buffer exist. |
| 2e. | fail() | | * To detect error. * ‘DetectWhat’.fail();   Example : cin.fail(); | --- |
| 2f. | clear() | | * To remove all values from a set. * ‘RemoveWhat’.clear();   Example : cin.clear(); |  |
| * In a loop, an infinite loop might happen because user input the data which do not match to the data type of the input should be accepted.   Example : cin b into A int data type’s variable inside a loop.  An infinite loop will happen because the cin cannot accept b as an input and it will become a buffer. At the next round of cin in the loop, the cin will auto accept the buffer which is the b but cin cannot accept b as an input and b will become a buffer again. This process will happen continuously inside the loop.   * cin.clear() and cin.ignore() can be applied together to avoid this situation happen. |
|  | | | | |
| 3a. | setw(x) |  | * **Header File :** #include <iomanip> * Used to output the value of an expression in a number of columns specific. * x is the number of columns(space). * The output is right justified. * If the number of columns(space) actual is more than x we set, the output automatically expands the x to the required number of columns(space) actual. |  |
| * **Other :** Limit The Amount Of Input Accepted * cin >> setw(‘Size’) >> ‘Variable’; * Variable’ must be an Array (C-String). * cin reads one less character than size set in setw(‘Size'). (Concept char Array - Refer 11 - b - 1a,1b) * Cannot accept whitespace such as cin.getline() and cin.get(). |
| 3b. | left |  | * Let the setw(x) output is left justified. |
| 3c. | right |  | * Let the setw(x) output is right justified. |
| 3d. | showpoint |  | * Causes a decimal point and trailing zeros to be displayed, even if there is no fractional part. (Show Significant Figures) * The default significant figures of decimal points are 6. It can be changed using setprecision(). * Once showpoint is applied on a certain function, other function also will be affected. |  |
| 3e. | fixed |  | * Displays floating-point numbers in fixed point notation. (Show Decimal Point) * The default floating-point numbers fixed by fixed are 6. It can be changed using setprecision(). * fixed has the higher order compare with showpoint. * Once fixed is applied on a certain function, other function also will be affected. |  |
| 3f. | setprecision(x) |  | * **Header File :** #include <iomanip> * Set the precision of floating-point numbers. * x is the precision of floating-point numbers required. (Significant Figures / Floating-Point Numbers) * setprecision(x) and showpoint apply together can set the significant figures required. * setprecision(x) and fixed apply together can set the floating-point numbers required. * fixed has the higher order compare with showpoint. * Once setprecision(x) is applied on a certain function, other function also will be affected. |  |

# Key Words (Selection & Loop)

1. **Selection :**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1a. | if |  | * To make logical choices. * It will process each condition one by one follow the sequence until reach the true condition. Only the statement in true condition will be processed. * if use on first condition.   else if use on the condition after first condition.  else use on last condition which is all the situations do not include in if and else if. Thus, there is no required to set condition for else. | --- |
| 1b. | else if |  | * If only one statement :   if (‘Condition’)  ‘Statement’;  else if (‘Condition’)  ‘Statement’;  else  ‘Statement’; |
| 1c. | else |  | * If more than one statement :   if (‘Condition’)  {  ‘Statement\_1’;  ↓  ‘Statement\_n’;  }  else if (‘Condition’)  {  ‘Statement\_1’;  ↓  ‘Statement\_n’;  }  else  {  ‘Statement\_1’;  ↓  ‘Statement\_n’;  } |
|  | | | | |
| 2. | switch |  | * To make logical choices. | --- |
| * switch (‘Expression’)   {  case ‘A’ : ‘Statement’;  break;  ↓  case ‘n’ : ‘Statement’;  break;  default : ‘Statement’;  break;  } |
| * If want to set a range :   case ‘From’ … ‘To’ : ‘Statement’;  break; |
| * If the break statement is not written, the program will continue execute the statement in next case until end which include the statement belong to default. * The data type of case can only be int, bool, char, constant. (Cannot be float, string type variables.) |

1. **Loop :**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1a. | while |  | * Is a Pre-Test Loop. |  |
| * while (‘Condition’)   {  ‘Repeated Action’;  } |
| 1b. | for |  | * Is a Pre-Test Loop. |  |
| * If only one statement :   for (‘Initialization’; ‘Condition’; ‘Updating’)  ‘Repeated Action’; |
| * If more than one statement :   for (‘Initialization’; ‘Condition’; ‘Updating’)  {  ‘Repeated Action’;  } |
| 1c. | do  while |  | * Is a Post-Test Loop. |  |
| * do   {  ‘Repeated Action’;  }  while (‘Condition’); |
|  | | | | |
| 2a. | break |  | * Can be used to terminate execution of a loop or switch statement. |  |
| * When used in an inner loop, terminates that loop only (inner loop) and returns to the outer loop.     When used in a switch statement, it can avoid the program to continue execute the statement in next case. |
| 2b. | continue |  | * Can be used to go to end of loop and prepare for next repetition. |  |
| 2c. | goto |  | * Can be used to translate connector symbols - jump to another part inside a program. |  |

# Key Words (Math Function)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1. | abs(x) |  | * **Header File :** #include <cmath> * Returns the absolute value of an integer. * If x is -5, abs(x) is 5. | --- |
| 2. | fabs(x) |  | * **Header File :** #include <cmath> * Returns the absolute value of its type of double argument. * Example : If x is -8.432, fabs(x) is 8.432. | --- |
| 3. | ceil(x) |  | * **Header File :** #include <cmath> * Returns the smallest integral value that is not less than x. * Example : If x is 45.23, ceil(x) is 46.0. | --- |
| 4. | floor(x) |  | * **Header File :** #include <cmath> * Returns the largest integral value that is not greater than x. * Example : If x is 45,23, floor(x) is 45.0. | --- |
| 5. | pow(x, y) |  | * **Header File :** #include <cmath> * Return xy. * If x is negative, y must be an integer. * If x is zero, y must be a positive integer. | --- |
| 6. | pow10(x) |  | * **Header File :** #include <cmath> * Return 10x. | --- |
| 7. | sqrt(x) |  | * **Header File :** #include <cmath> * Return the positive square root of x. (x >= 0) | --- |
| 8. | exp(x) |  | * **Header File :** #include <cmath> * Returns ex. | --- |
| 9. | log(x) |  | * **Header File :** #include <cmath> * Returns the natural logarithm. * Actually, it is ln(x). Base-e logarithm. | --- |
| 10. | log10(x) |  | * **Header File :** #include <cmath> * Returns the base-10 logarithm. | --- |
| 11. | sin(x) |  | * **Header File :** #include <cmath> * Sine. | --- |
| 12. | cos(x) |  | * **Header File :** #include <cmath> * Cosine. | --- |
| 13. | tan(x) |  | * **Header File :** #include <cmath> * Tangent. | --- |

# Key Words (Time Function)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1. | time() |  | * **Header File :** #include <ctime> * Returns current calendar time. | Refer 9 - 1,2 |

# Key Words (Data Conversion Function)

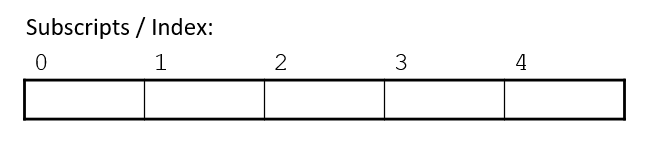
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1. | rand() |  | * **Header File :** #include <cstdlib> * Returns a random integer between 0 and the largest int the computer holds. * If not define any seed for rand(), the result of rand() will always same. * Same seed will also produce same random number. Thus, we need to always change the seed to produce a random number. * Determine the range of the random number : * [0, x) or [0, x-1] : rand()%‘x’; * [y, x+y) or [y, x+y-1] : rand()%‘x’+‘y’; |  |
| 2. | srand() |  | * **Header File :** #include <cstdlib> * Initializes random number generator. (Use to define the seed.) * We can use time() as the seed because the calendar time of the computer system is always changing. |
| 3. | atoi() |  | * **Header File :** #include <cstdlib> * Converts C-string to an int value, returns the value. |  |
| 4. | atoll() |  | * **Header File :** #include <cstdlib> * Converts C-string to a long value (int), returns the value. |
| 5. | atof() |  | * **Header File :** #include <cstdlib> * Converts C-string to a double value, returns the value. |
| 6. | exit() |  | * **Header File :** #include <cstdlib> * Terminates the execution of a program. * exit(0); |  |

# Key Words (Character Classification and Conversion Function)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1. | isalpha() |  | * **Header File :** #include <cctype> * True if arg. is a letter, false otherwise. | --- |
| 2. | isalnum() |  | * **Header File :** #include <cctype> * True if arg. is a letter or digit, false otherwise. | --- |
| 3. | isdigit() |  | * **Header File :** #include <cctype> * True if arg. is a digit 0-9, false otherwise. | --- |
| 4. | islower() |  | * **Header File :** #include <cctype> * True if arg. is a lowercase letter, false otherwise. | --- |
| 5. | isprint() |  | * **Header File :** #include <cctype> * True if arg. is a printable character, false otherwise. | --- |
| 6. | ispunct() |  | * **Header File :** #include <cctype> * True if arg. is a punctuation character, false otherwise. | --- |
| 7. | insupper() |  | * **Header File :** #include <cctype> * True if arg. is an uppercase letter, false otherwise. | --- |
| 8. | isspace() |  | * **Header File :** #include <cctype> * True if arg. is a whitespace character, false otherwise. | --- |
| 9. | toupper() |  | * **Header File :** #include <cctype> * Return uppercase equivalent if char argument is lowercase letter; otherwise, return input unchanged. |  |
| 10. | tolower() |  | * **Header File :** #include <cctype> * Return lowercase equivalent if char argument is uppercase letter; otherwise, return input unchanged. |  |

# Array

1. **Array :** Variable that can store a collection of data of the same type.



* ‘DataType’ ‘Variable’[‘n’];
* The last element’s subscript is n-1 where n is the number of elements in the array (Subscript / Index).

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Array** | **Explain** | **Example** |
| 1a. | Array Initialization | * The initialization list cannot exceed the array size.      * The size of each element will be based on the size of its data type (int = 4 bytes). * The size of tests will be the total size of all elements (12 bytes). |  |
| 1b. | Array Initialization (char) | * Both of these are same :      * Must leave room for \0 at end of array because each string will have a \0 or 0 at the end of the string. |
| 1c. | Partial Array Initialization | * If array is initialized with fewer initial values than the size declarator, the remaining elements for int data type will be set to 0; for char data type will be NULL. |
| 1d. | Implicit Array Sizing | * Can determine array size by the size of the initialization list. |
| 1e. | No Bounds Checking in C++ | * The subscripts that are beyond the bounds of the array can be used. * Be careful not to use invalid subscripts. * Doing so can corrupt other memory locations, crash programs or lock up computers and cause elusive bugs. |
|  | | | |
| 2a. | Array Assignment | * ‘Variable 1’ = ‘Variable 2’; * It cannot be applied on the array. * Even it is a char array : |  |
| * If a variable is an array, assign element-by-element : |
| 2b. | Array Printing | * You can output the contents of a variable with data type char directly :     But this only works with variable with data type char. |  |
| * If not variable with data type char, output element-by-element : |
|  | | | |
| 3. | Parallel Array | * Two or more arrays that contain related data. |  |
|  | | | |
| 4. | Function Argument | * To pass an array to a function, just use the array name.     To define a function that takes an array parameter, use empty [ ] for array argument:     * When passing an array to a function, it is common to pass array size so that function knows how many elements to process. * CAUTION : Array names in functions are like Reference Variables – changes made to array in a function are reflected in actual array in calling function. |  |

1. **Dimensional Array :**

* Can define one array for multiple sets of data.

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Dimensional Array** | **Explain** | **Example** |
| 1a. | Two-Dimensional Array | * Like a table in a spreadsheet. * Use two size declarators in definition :   int ‘Variable’[‘Rows’][‘Column’];   * Example : |  |
| * **Other :** Two-Dimensional Array Initialization * Two-Dimentional Arrays are initialized row-by-row. |
| * **Other :** Two-Dimensional Array Function Argument * Multidimensional Array must have bounds for all dimensions except the first.   Example :    Use empty [ ] for row, size declarator for column in prototype and header. |
| 2a. | More Dimensional Array | * Can define arrays with any number of dimensions.   Example :  short ‘FunctionName’[ ][‘Size 2D’][‘Size 3D’]; // Three-Dimensional Array  double ‘FunctionName’[ ][‘Size 2D’][‘Size 3D’][‘Size 4D’]; // Four-Dimensional Array | --- |
| * When used as parameter, specify all except first dimension in prototype and heading.   Example :  void ‘FunctionName’ (short [ ][‘Size 2D’][‘Size 3D’]); // Three-Dimensional Array (Prototype)  void ‘FunctionName’ (short ‘Variable’[ ][‘Size 2D’][‘Size 3D’]) // Three-Dimensional Array (Header) |

# Key Words (File Operation)

1. **Steps of File Operation :**
2. Data type declaration.
3. Open the file.
4. Use the file (Read From, Write To or Both).
5. Close the file.

* If not, it may limit the number of open files or buffered output data waiting to be sent to the file.

1. **Key Words (File Operation - Declaration) :**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1. | ifstream |  | * **Header File :** #include <fstream> * Data type for input files. * Use to read content from a file. * ifstream ‘FileVariable’; * Open fails if file does not exist. |  |
| 2. | ofstream |  | * **Header File :** #include <fstream> * Data type for output files. * Use to write something into a file. * ofstream ‘FileVariable’; * File created if no file exists. * File contents erased if file exists. |  |
| 3. | fstream |  | * **Header File :** #include <fstream> * Data type for both input and output files. * fstream ‘FileVariable’; * Must specify mode on the open() statement : * Input Mode : ios::in * Output Mode : ios::out * Binary Mode : ios::binary * Append Mode : ios::app |  |
| * If ios::in and ios::out apply together, ios::in will have the higher order which it will open fail and not auto create when file does not exist. |

1. **Key Words (File Operation - Open) :**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1. | open() |  | * Create a link between file name (outside the program) and file stream object (inside the program). * Filename may include drive and / or path info. * ifstream / ofstream : ‘FileVariable’.open(“FileName”); * fstream (Only 1 Modes) : ‘FileVariable’.open(“FileName”, ‘Modes’);   fstream (More Than 1 Modes) : ‘FileVariable’.open(“FileName”, ‘Modes’ | ‘Modes’); | Refer 12 - b - 1,2,3 |
| * **Other :** Open File At Declaration * ifstream : ifstream ‘FileVariable’ (“FileName”); * ofstream : ofstream ‘FileVariable’ (“FileName”); * fstream (Only 1 Modes) : fstream ‘FileVariable’ (“FileName”, ‘Modes’);   fstream (More Than 1 Modes) : fstream ‘FileVariable’ (“FileName”, ‘Modes’ | ‘Modes’); |  |
| 2. | is\_open() |  | * Check if a file is open. * ‘FileVariables’.is\_open()   Return 1 if file is opened.  Return 0 if file is closed. |  |
| * Example :   if ( !‘FileVariables’.is\_open() )  {  cout << “ERROR: Cannot open file.” << endl;  exit(1);  } |
| 3. | fail() |  | * To detect error. (Refer 5 - 2e) * ‘DetectWhat’.fail(); * Return 1 if file open error (File Not Exist). * Return 0 if file open without error (File Exist). |  |
| * Example :   if ( ‘FileVariables’.fail() )  {  cout << “ERROR: Cannot open file.” << endl;  exit(1);  } |

1. **Key Words (File Operation - Use) :**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1. | **(FILE DATA INPUT AND OUTPUT)** |  | * Can use << to send data to a file. * ‘FileVariable’ << ‘Variable / String Literal’; * Can use >> to copy data from file to Variable. * ‘FIleVariable’ >> ‘Variable’; |  |
| * Use while and getline() when accept a string from file with whitespace or ENTER(endl). |
| 2. | eof() |  | * Test for end of input file. * Return 1 if the end of the file has been reached. * Return 0 if the end of the file has not been reached. |  |
| * Use while and eof() when accept a string from file with whitespace or ENTER(endl). |

1. **Key Words (File Operation – Close) :**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1. | close() |  | * Use to close a file. * ‘FIleVariables’.close(); | Refer 12 - b - 1,2,3  Refer 12 - c - 1,2,3  Refer 12 - d - 1,2 |
| 2. | remove() |  | * Use to delete the file. * remove(‘FileVariable’); | --- |

# Pointers

1. **Address / Pointer Variable :**

* Address : A uniquely defined memory location which is assigned to a variable.
* Pointer Variable : A variable that holds an address.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Operator Name** | **Operator** | **Explain** | **Example** |
| 1. | Address Operator | & | * Used to get the address of a variable. (Hexadecimal Form) * &’Variable’ |  |
| 2. | Indirection Operator | \* | * Used to set a variable as Pointer Variable. (Inside Declaration Part) * ‘DataType’ \*’Variable’; * The Pointer Variable must be declared together with the variable declaration.      * The address of Pointer Variable hold can be assigned inside or outside the variable declaration.      * The data type of Pointer Variable and data type of variable it points to should be same. (Cannot Mix Data Type)      * Can test for an invalid address. |  |
| * Used to access the item that the pointer points to. (Outside Declaration Part) * \*’Variable’ * \*&’Variable’ is equal to ‘Variable’. |
| * Any change of the data of \*’Variable’ will also affect the data of the variable it points to. |

1. **Pointers & Array :**

* Example :

A screenshot of a computer program

Description automatically generated

1. **Pointers & Array (Arithmetic) :**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Operation** | **Operator** | **Explain** | **Example** |
| 1. | Unary Operator | ++, -- | * Postfix Expression / Prefix Expression * Increment / Decrement * Example : \*(‘Pointer’++) |  |
| 2. | Arithmetic Operator | +, - | * Plus / Minus * Example : \*(‘Pointer’ + ‘Integer’) |
| 3. | Combined Assignment Operator | +=, -= | * Example : \*(‘Pointer’ += ‘Integer’) |
| 4. | **(Addition and Subtraction of The Address)** |  | * When we add or minus a N value (integer) to an address of the Pointer, it will move forward or backward by (N \* ‘DataType’s Size’) bytes. * Example : | --- |
| 5. | Minus | - | * Find the distance between both Pointer. * Both Pointer must be same data type. * Both Pointer should be in a same array.   (If not, find the distance between two Pointer is not meaningful since the memory address assigned by the system is random and not continuous.) |  |
| * ‘Pointer1’ - ‘Pointer2’ * Return (+) N integer if the address ‘Pointer1’ has higher order compare with ‘Pointer2’ in an array. * Return (-) N integer if the address ‘Pointer1’ has lower order compare with ‘Pointer2’ in an array. * N = (‘Address Pointer1’ - ‘Address Pointer2’) / ‘DataType of Pointer’ * N is the distance (Amount of ‘Size in Bytes of Array’s DataType’) between both Pointer. |

1. **Pointer & Function Parameters :**

* A computer screen with white text and blue letters

  Description automatically generated

1. **Pointer & Constant :**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Types** | **Explain** | **Example** |
| 1. | Constant To Pointer | * const ‘DataType’ \*‘PointerVariable’ = &‘Variable’; * Data of \*‘PointerVariable’ cannot be changed. * ‘Variable’ of ‘PointerVariable’ points to can be changed. |  |
| 2. | Constant Pointer | * ‘DataType’ \* const ‘PointerVariable’ = &‘Variable’; * Data of ‘PointerVariable’ can be changed. * ‘Variable’ of ‘PointerVariable’ points to cannot be changed. |  |
| 3. | Constant Pointer To Constant | * const ‘DataType’ \* const ‘PointerVariable’ = &‘Variable’; * Data of ‘PointerVariable’ cannot be changed. * ‘Variable’ of ‘PointerVariable’ points to cannot be changed. |  |

1. **Dynamic Memory Allocation :**

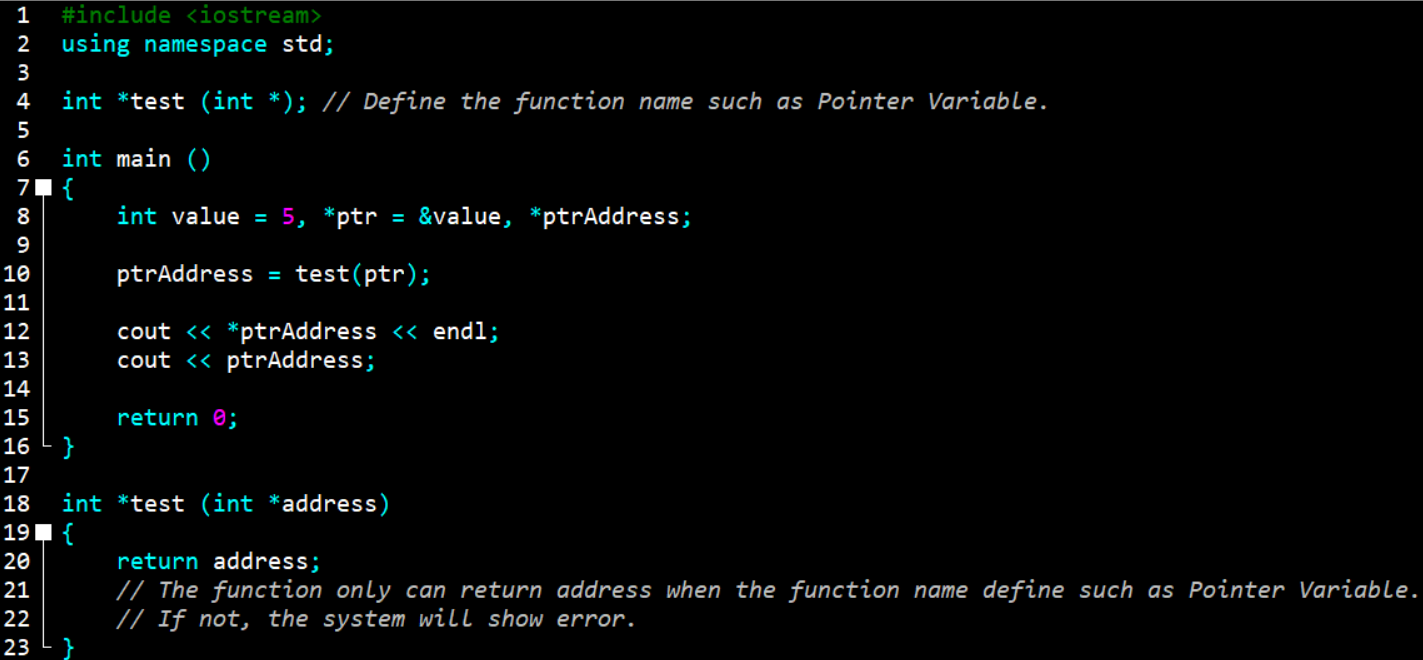
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Key Words** | **Bit Width** | **Explain** | **Example** |
| 1. | new |  | * Can allocate storage for a variable while program is running. * Computer returns address of newly allocated variable. |  |
| * For Non-Array :   ‘DataType’ \*‘PointerVariable’ = new ‘DataType’;   * For non-array case, we apply this is because we can reduce the burden of the program although it is very little. * If we apply ‘DataType’ ‘Variable’; as usual, the program will always execute it until the end. * If we apply new, the program will only execute it when we need,and we can use delete to delete it. Thus, the program has no need to execute it until the end. |
| * For Array :   ‘DataType’ \*‘PointerVariable’ = new ‘DataType’[‘Size’]; |
| 2. | delete |  | * Use to free dynamic memory. |
| * For Non-Array :   delete ‘PointerVariable’; |
| * For Array :   delete [ ] ‘PointerVariable’; |

1. **Pointer & Function Return Address :**

* Pointer only can be returned by the type of the function :

‘DataType’ \*‘FunctionName’(); // Prototype

* The function must not return a pointer to a local variable in the function.
* A function should only return a pointer :
* to data that was passed to the function as an argument, or
* to dynamically allocated memory.
* Example :



# Structured Data

1. **Structure :**

* C++ construct that allows multiple variables to be grouped together.

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Struct** | **Explain** | **Example** |
| 1a. | struct Declaration | * struct ‘structName’   {  ‘DataType’ ‘FieldName1’;  ‘DataType’ ‘FieldName2’;  ↓  ‘DataType’ ‘FieldName\_n’;  };   * struct names commonly begin with uppercase letter. * struct declaration does not allocate memory or create variables. |  |
| 1b. | struct Variable Declaration | * ‘structName’ ‘structVariable’; * ‘structVariable’ in the function will become the variable under the struct data type of ‘structName’ which is able to access it. |
| 1c. | Accessing Structure Members | * ‘structVariable’.‘FieldName’ |
| 1d. | Initializing Structure | * Initializing At Declaration :   ‘structName’ ‘structVariable’ = {‘DataField1’, ‘DataField2’, … , ‘DataField\_n’}; |
| * Initializing By Member :   ‘structVariable’.‘FieldName1’ = ‘DataField1’;  ‘structVariable’.‘FieldName2’ = ‘DataField2’;  ↓  ‘structVariable’.‘FieldName\_n’ = ‘DataField\_n’; |
|  | | | |
| 2. | Nested Structure | * A structure can contain another structure as a member. |  |

1. **Array of Structure :**

* Example :

A computer screen shot of code

Description automatically generated

1. **Structure & Function :**

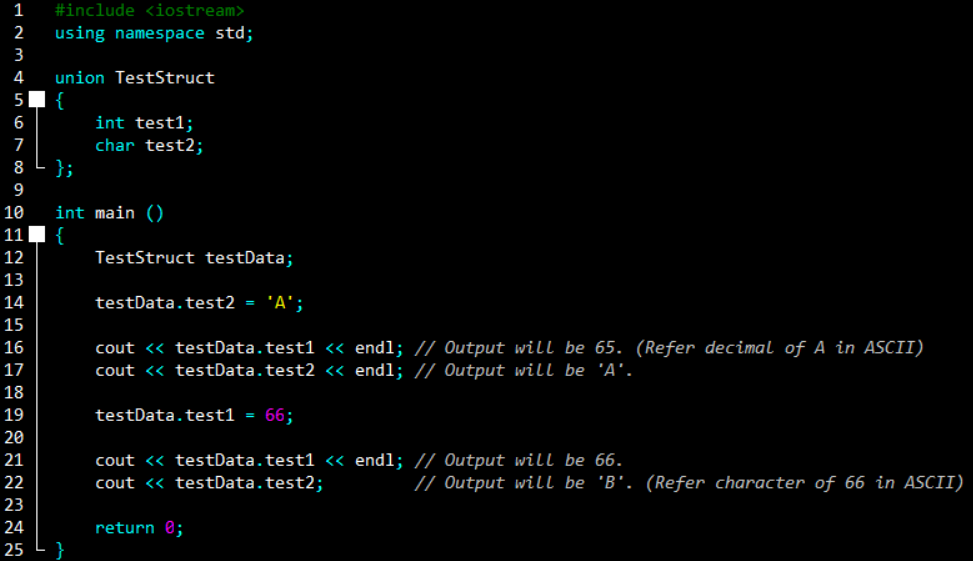
|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Function** | **Explain** | **Example** |
| 1. | Function Argument (struct) | * Pass Entire Struct Variables to Functions :   ‘FunctionType’ ‘FunctionName’ (‘structName’); // Prototype  ‘FunctionName’(‘structVariable’); // Calling  ‘FunctionType’ ‘FunctionName’ (‘structName’ ‘structVariable’); // Header   * The function can access the whole data inside the ‘structVariable’. |  |
| * Pass Members of Struct Variables to Functions :   ‘FunctionType’ ‘FunctionName’ (‘DataType FieldName1’); // Prototype  ‘FunctionName’(‘structVariable’.‘FieldName1’); // Calling  ‘FunctionType’ ‘FunctionName’ (DataType FieldName1’ ‘Variable’); // Header   * The function only can access the data of ‘FieldName1’ in ‘structVariable’. |
|  | | | |
| 2. | Function Return (struct) | * ‘FunctionType’ must be ‘structName’. |  |

1. **Structure & Pointer :**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Pointer** | **Explain** | **Example** |
| 1. | Pointer Declaration (struct) | * A structure variable has an address. * structName’ \*‘Pointer\_structVariable’ = &‘structVariable’; | A computer screen with white text and colorful letters  Description automatically generated |
| 2. | Access Item of Pointer Point To (struct) | * Apply Dereference :   (\*‘structVariable’).‘FieldName’   * Must use () to dereference pointer variable. |
| * Clearer Notation :   ‘structVariable’->‘FieldName’   * Can use structure pointer operator to eliminate (). |

1. **Unions :**

* Like a struct but replace the struct with union.
* All members share a single memory location and only one member of the union can be used at a time.
* All members of the union will share same data.
* If used more than one member of the union, the latest write in data will overwrite all the data write in before.
* Example :



1. **Enumerated Data Types :**
2. An Enumerated Data Type is a programmer-defined data type.
3. It consists of values known as Enumerators, which represent integer constants.

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Enumerated** | **Explain** | **Example** |
| 1. | Enumerated Data Types Declaration | * enum ‘Enumerated Data Types Name’   {  ‘Enumerator1’, ‘Enumerator2’, … , ‘Enumerator\_n’  };   * The Enumerator is not string, so they are not enclosed in quotes. They are Identifiers. * Enumerators must be unique within the same scope. * Enumerators cannot declare with same name although in difference Enumerated Data Type. |  |
| * Enumerator can think of it as integer named constant. * Internally, the compiler assigns integer values to the Enumerators, beginning at 0. * ‘Enumerator1’ == 0   ‘Enumerator2’ == 1  ↓  ‘Enumerator\_n’ == ‘n’ - 1 |
| 2. | Enumerator Variable Declaration | * Declare Inside Enumerated Data Types Declaration :   enum ‘Enumerated Data Types Name’  {  ‘Enumerator1’, ‘Enumerator2’, … , ‘Enumerator\_n’  }  ‘enumVariable’; |
| * Declare Outside Enumerated Data Types Declaration :   ‘Enumerated Data Types Name’ ‘enumVariable’; |
| 3. | Enumerator Variable Assignment | * Assign With Enumerator :   ‘enumVariable’ = ‘Enumerator\_n’; |
| * Assign With Integer :   ‘enumVariable’ = static\_cast <‘Enumerated Data Types Name’> (‘Integer’); |
|  | | | |
| 4. | Anonymous Enumerated Types | * Enumerated Data Type without name. * enum   {  ‘Enumerator1’, ‘Enumerator2’, … , ‘Enumerator\_n’  }; | --- |