C++ LANGUAGE CONSTRUCTS

CHAPTER 3

KABIN DEVKOTA

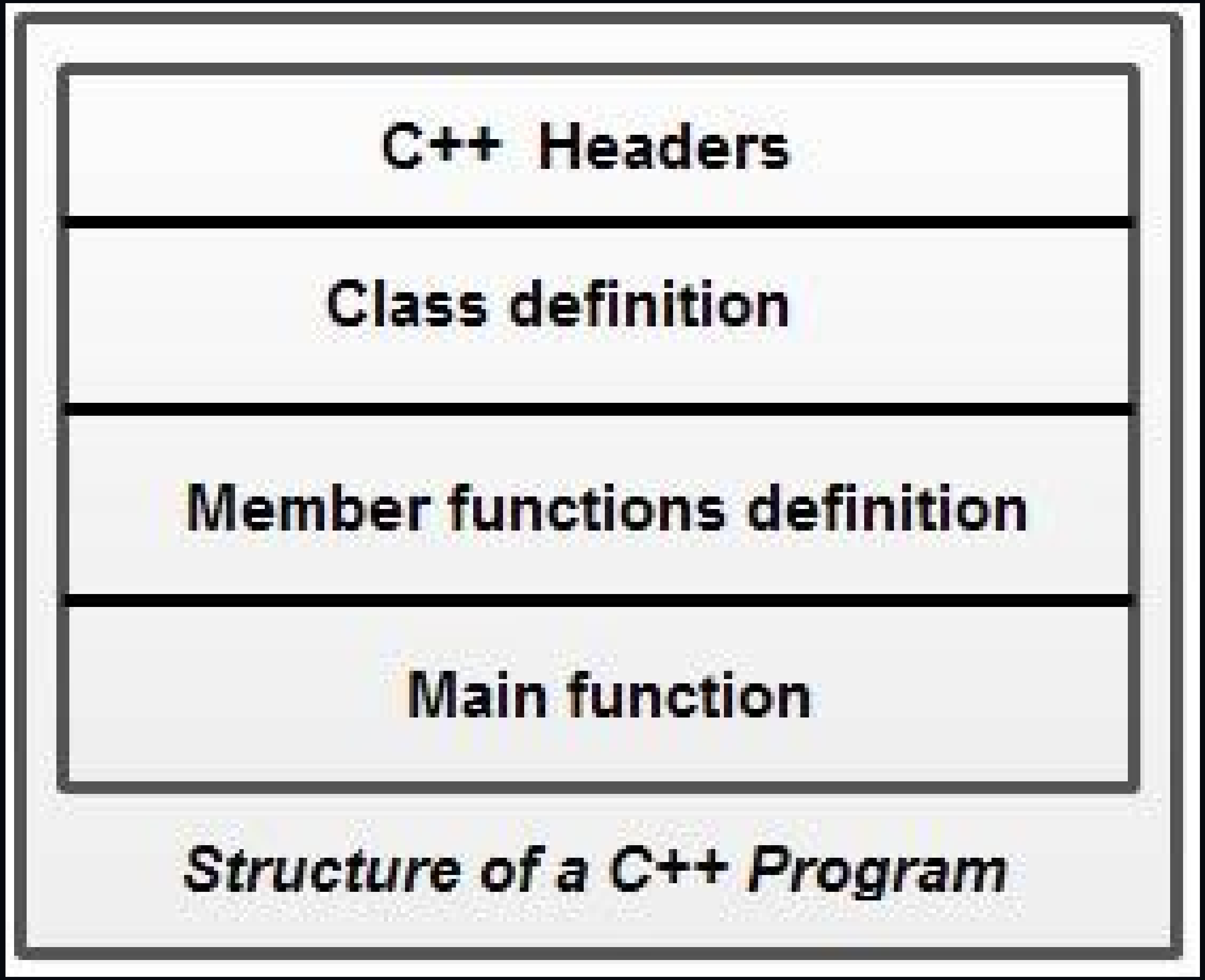
UDITA BISTA

TABLE OF CONTENTS

Our content is
divided into the given topics.
Each part will be described in the slides
that follow

C++ PROGRAM STRUCTURE

- Following figure explains the structure of C++ program. C++ commonly organize the program into three sections.



Structure of a C++ Program

CHARACTER SETS AND TOKENS

Character Sets:

- Character set is a set of valid characters that a language can recognize.
- A character represents any letter, digit, or any other sign.
- The C++ has the following character set.
  - Digit: 0-9
  - Letters:A -Z, a-z
  - White spaces: Blank space, Horizontal tab, Carriage return, Newline, Form feed
  - Special Symbols: space + * / \ % ^ ( ) { } [ ] = > < , ‘ “ $ & | ! ~ : ? _ # <= >= ==
  - Other characters: C++ can process any of the 256 ASCII characters as data or as Literals

CHARACTER SETS AND TOKENS

Tokens:

- Token is the smallest unit of a program. C++ has the following tokens:

1. Keywords
2. Identifiers
3. Constants
4. Operators
5. Punctuators

Tokens:

1. **Keywords:** They are reserved words and cannot be used for any other purposes. All 32 keywords from ANSI C are also valid in C++. C++ has additional keywords. We will discuss additional keywords in Chapter 4.

2. **Identifiers:** They are programmer-defined elements like the names of the variables, functions, arrays, structures, etc. These are tokens which are sequence of letters, digits and underscore(_). Identifiers are used to give unique names to the elements in the program. Both uppercase and lowercase letters are permitted, C++ is case sensitive and it treats upper and lowercase characters differently. The rules of defining identifier:

    a. The first character must be an alphabet (or underscore)
    b. Must consist of only letters, digits, or underscore
    c. Cannot use a keyword
    d. Must not contain whitespace or special character

    The following are some valid identifiers:

    Sagarmatha, KEC, Data12, _Rise, _Do12, _ for, _foo_bar

Tokens:

3. **Constants:** Constants, also called literals, are the actual representation of the values used in a program. They never change their value during the program run

   a. Integer constant
   b. Character constant
   c. Floating constant
   d. String constant

4. **Punctuators:** The special symbols which act like punctuation marks are called punctuators. The following characters are used as punctuators (also known as as separators) in C++: [ ] ( ) { } , ; : * … = # That enhances a program’s readability.

Tokens:

5. **Operators:** Operators are special symbols used to perform mathematical or logical operations. There are mainly following types of operators in C++:

   a. Arithmetic operator +, , *, /, %
   b. Increment/Decrement operator ++, --
   c. Relational operator <, <=, >, >=, ==, !=
   d. Logical operator && , ||, !
   e. Assignment operator =
   f. Bitwise operator &, |, ^
   g. Conditional operator ? :

      exp1 ? exp2 : exp3

      large = (a>b) ? a : b;

   h. Other operator: comma (,), sizeof(), pointer-operator(& and *), selection--operator(. and ->)

VARIABLE DECLARATION AND EXPRESSION

Variable Declaration

Variables are entities which hold values and whose value may change throughout program execution. Each and every variable must be declared beforehand.

data type variable_name1, variable_name 2,.......variable_nameN;

```
E.g.  int length, breadth;
      float  area;
      char  section;
```

Also, variables can be initialized during declaration as:

```
int length = 5, breadth = 3;
float area = 14.57;
char section = ‘A’;
```

VARIABLE DECLARATION AND EXPRESSION

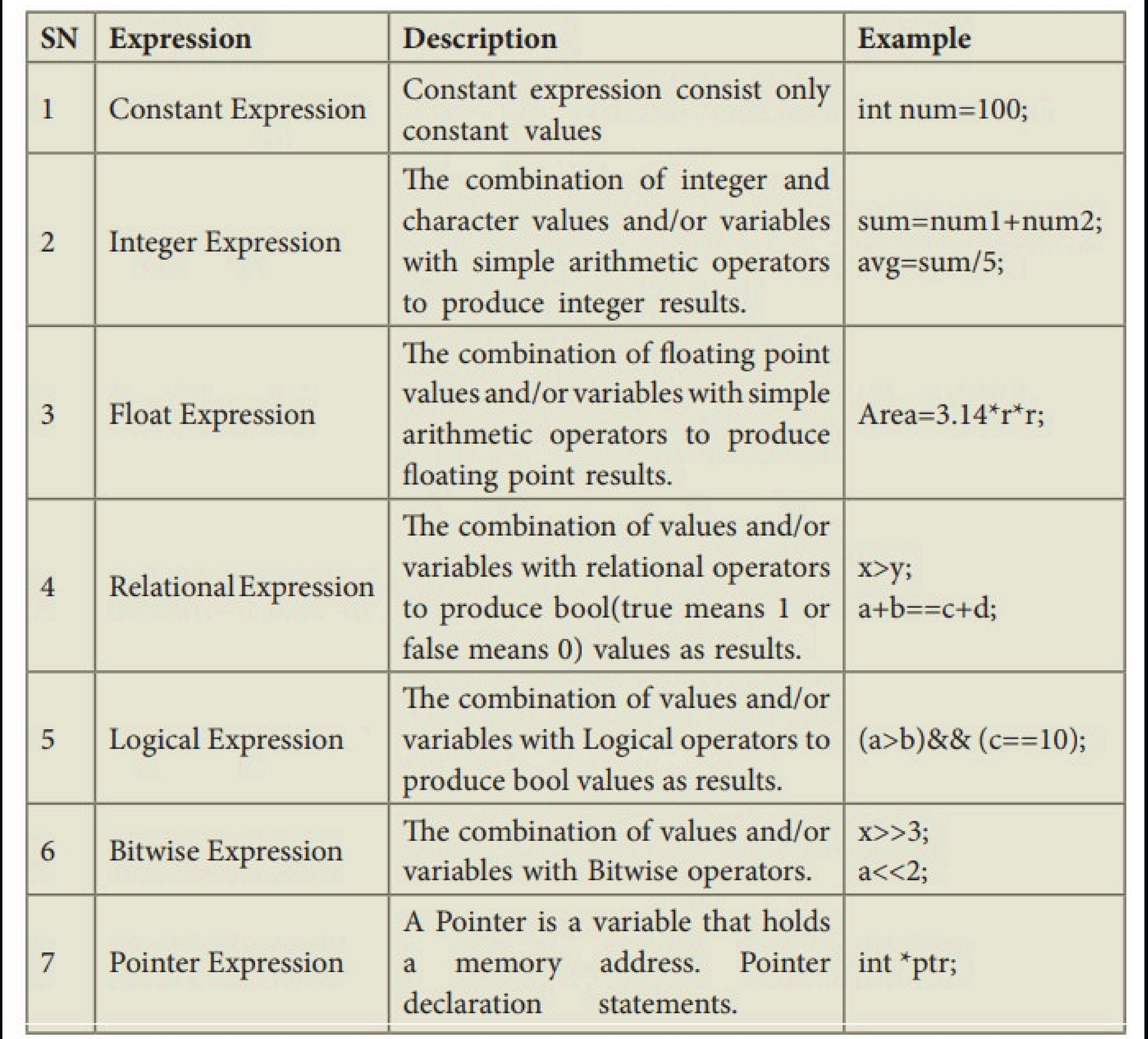
Expression

Any arrangement of variables, constants and operators that specifies a computation is called an expression. Thus, ***alpha + 12*** and ***(alpha -37)*beta/2*** are expressions.
Parts of expressions may also be expressions. In the second example, alpha-37 and beta/2 are both expressions.

Note: The expressions aren't the same as statements. The statements terminate with a semicolon. There can be several expressions in a statement.

Types of expressions (Figure from insights page 13)

Types of Expression



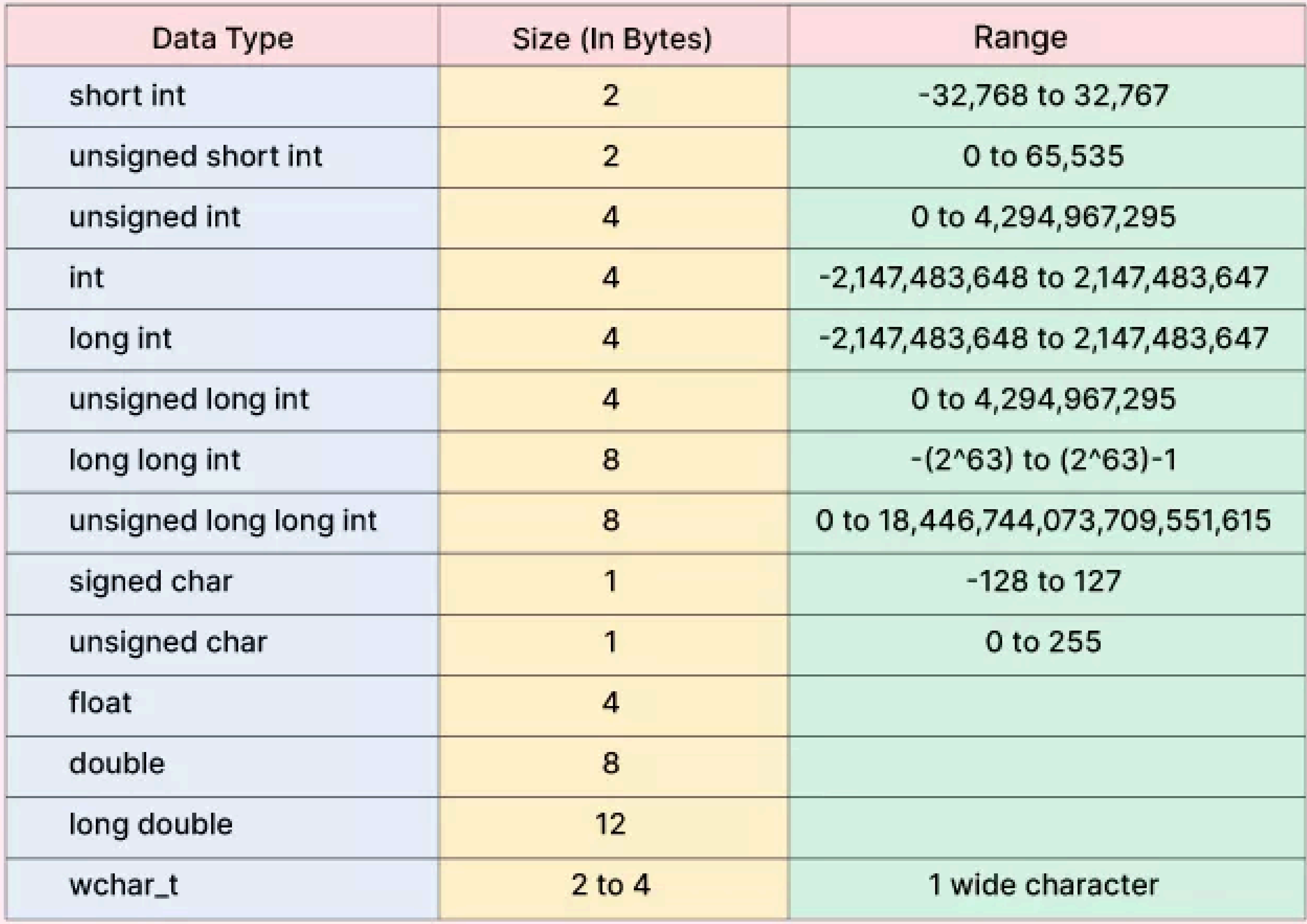
SN	Expression	Description	Example
1	Constant Expression	Constant expression consist only constant values	int num=100;
2	Integer Expression	The combination of integer and character values and/or variables with simple arithmetic operators to produce integer results.	sum=num1+num2; avg=sum/5;
3	Float Expression	The combination of floating point values and/or variables with simple arithmetic operators to produce floating point results.	Area=3.14*r*r;
4	Relational Expression	The combination of values and/or variables with relational operators to produce bool(true means 1 or false means 0) values as results.	x>y; a+b==c+d;
5	Logical Expression	The combination of values and/or variables with Logical operators to produce bool values as results.	(a>b)&& (c==10);
6	Bitwise Expression	The combination of values and/or variables with Bitwise operators.	x>>3; a<<2;
7	Pointer Expression	A Pointer is a variable that holds a memory address. Pointer declaration statements.	int *ptr;

STATEMENTS

- Statements are the instructions given to the computer to perform any kind of action, be it evaluating expressions or making decisions or repeating actions.
- A statement forms the smallest executable unit within a C++program.

```
Sum = a + b;
```

DATA TYPES



Data Type	Size (In Bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	
wchar_t	2 to 4	1 wide character

C++ DATA TYPES

1. Built-in type
    - Void
    - Integral type
        i. int
        ii. char
    - Floating type
        iii. float
        iv. double

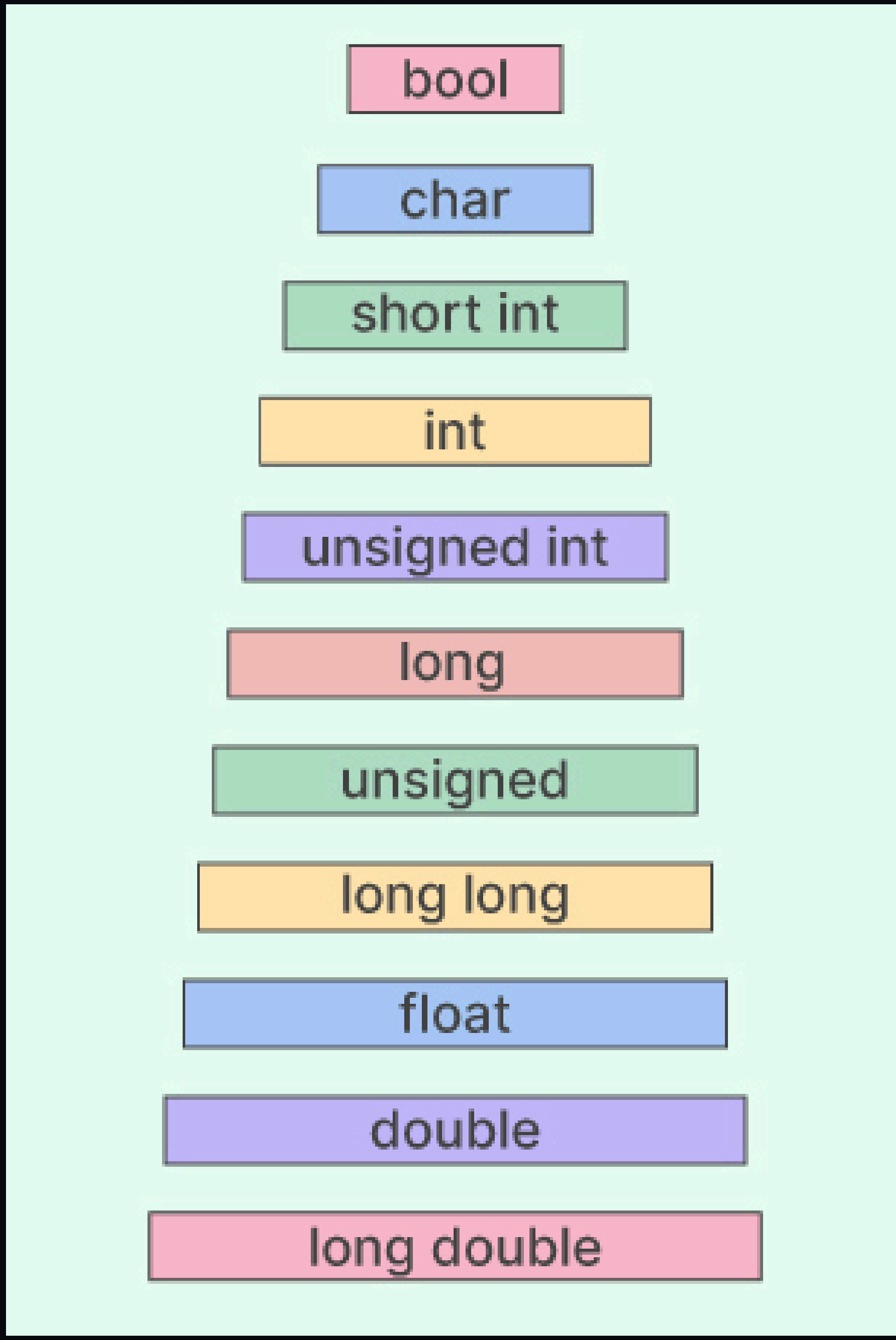
2. Derived Type
    - Array
    - Pointer
    - Reference

3. User defined type
    - Structure
    - Union
    - Class
    - Enumeration

TYPE CONVERSION AND PROMOTION RULES

Implicit Type Conversion

- Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is called implicit or automatic type conversion.
- For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the “smaller” type is converted to the “wider” type.



TYPE CONVERSION AND PROMOTION RULES

Explicit Type Conversion

C++ permits explicit type conversion of variables using the type cast operator. Hence, also called the 'cast'. This conversion is done by programmers as per need.

typename (expression)

Eg. Average = sum / float (i);

ANSI C++ adds the following new cast operators

- const_cast
- static_cast
- dynamic _cast
- reinterpret_cast

PREPROCESSOR DIRECTIVES

- The lines beginning with hash (#) sign are called preprocessor directive.
- A preprocessor directive is an instruction to the compiler(they are not program statements)
- A part of the compiler called the preprocessor deals with these directives before it begins the real compilation process.

E.g.,

```
#include<iostream>
#define PI 3.1415  etc
```

NOTE: They don't terminate with a semicolon

NAMESPACE

The namespace is used for the logical grouping of program elements like variables, classes, functions, etc. If some program elements are related to each other, they can be put into a single namespace. The namespace helps to localize the name of identifiers so that there is no naming conflict across different modules designed by different members of programming team.

Syntax for defining namespace is as follows:

```
namespace namespace_name
{
 //declaration of variables, functions,classes etc
} // no semicolon at the end
```

Example:

```
namespace myNamespace
{
 int a, b;
 void name();
}
```

NAMESPACE

Example:

```
namespace myNamespace
{
 int a, b;
 void name();
}
```

In this case, a and b are normal variables and name() is a user-defined function declared within a namespace called myNamespace. In order to access these variables and user-defined functions from outside the myNamespace namespace, we have to use the scope resolution operator (: :).For example, to access the previous variables and function from outside myNamespace we can write:

1. myNamespace::a
2. myNamespace::b
3. void myNamespace::name()

NAMESPACE

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing ambiguity (redefinition) errors. For example:

```
// namespaces
#include <iostream>
using namespace std;
namespace first
{
int var = 5;
}
namespace second
{
double var = 3.1416;
}
int main () {
cout<< first::var<< endl;
cout << second::var << endl;
return 0;
}
```

Output:

```
5
3.1416
```

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second. No redefinition errors happen thanks to namespaces.

NAMESPACE

Example:

```
#include<iostream>
 using namespace std;
namespace Rectangle
 {
int width;
 int height;
void area();
}
```

```
int main()
 {
Rectangle::width = 10;
Rectangle::height = 2;
Rectangle::area();
return 0;
 }
void Rectangle::area()
{
cout << "\nThe Area is: " << width*height << endl;
}
```

Output:

```
The Area is: 20
```

USER DEFINED CONSTANT CONST

The keyword ***const***(for constant) is used to specify that the value of the variable will not change throughout the program. Any attempt to alter the value of the variable defined with this qualifier will result in an error message from the compiler. Variables with this qualifier are often named in all uppercase, as a reminder that they are constants.

Syntax:

const [data_type] variable_name = const_value;

```
Eg: const int TRUE = 1;
       const float VAR = 3.2;
```

USER DEFINED CONSTANT CONST

Program Example:

```
#include <iostream.h>
#include <conio.h>
int main( )
{
clrscr( );
float r, a ;
const float PI=3.14 ;
cout<<“Enter r:”<< endl ;
cin>>r ;
a = PI*r*r ;
cout<<endl<< “Area of circle=”<<a ;
getch( );
}
```

In the program, if we modify a constant type variable, then it leads to the compilation error: cannot modify a const object.

INPUT/OUTPUT STREAMS AND MANIPULATORS

Input stream

Input stream is a sequence of characters from input device to the computer. The input stream allows us to perform read operations from the input devices such as keyboard, disk etc.
The input operation is performed using the ‘***cin***’ object.
It uses an extraction operator(>>) that extracts the value from the keyboard and inserts it to the variable on its right side.
For eg:
 cin >> varname; //cin extracts value from the keyboard and assigns it to variable ‘varname’.

INPUT/OUTPUT STREAMS AND MANIPULATORS

Output stream

Output stream is a sequence of characters from the computer to the output device. The ouput stream allows us to perform write operations on standard output devices.
The output operation is performed using the ‘***cout***’ object.
It uses an insertion operator(<<) that inserts the contents of the variable on its right to the object on its left.
For eg:
 cout << varname; //cout inserts the value of the variable ‘varname’ to the output device.

INPUT/OUTPUT STREAMS AND MANIPULATORS

Manipulators

Manipulators are the most common way to control output formatting. They are used to format the data display. In C++, the functions for formatting output are included under the <iomanip> header file.
Some of the manipulators are discussed below:
1) ***endl :*** Write a newline ('\n') and flush buffer.
2) ***setw(n) :*** Sets minimum field width on output. This sets the minimum size of the field a larger number will use more columns. To print a column of right justified numbers in a seven-column field:

```
n = 9;
cout << setw(7) << n << endl;
Output : ......9
```

INPUT/OUTPUT STREAMS AND MANIPULATORS

Manipulators

3) setfill(ch) : Only useful after setw(). If a value does not entirely fill a field, the character ch will be used to fill in the other characters. Default value is blank. Same effects as cout.fill(ch) For example, to print a number in a 4 character field with leading zeros (eg, 0007):

```
n = 8;
cout << setw(6) << setfill(‘0’) <<n << endl;
Output : 000008
```

DYNAMIC MEMORY ALLOCATION WITH NEW AND DELETE

Allocation of memory during runtime is known as dynamic memory allocation(DMA). C++ provides two operators for DMA new and delete.

New Operator:
C++ provides a new approach to obtaining blocks of memory using the new operator. This operator obtains memory from the operating system and returns a pointer to its starting point.

Delete Operator:
If our program reserves many chunks of memory using new, eventually all the available memory will be reserved and the system will crash. To ensure safe and efficient use of memory, the delete deallocates the memory pointed by the given pointer.

DYNAMIC MEMORY ALLOCATION WITH NEW AND DELETE

The general form of using new and delete is as follows:

datatype *pointervariable;
pointervariable = new datatype; //allocates single variable
pointervariable = new datatype[size]; //allocates an array of size elements
delete pointervariable; //if memory was allocated for a single variable
delete [ ] pointervariable; //alternatively when memory is allocated for an array

We can also initialize the the memory value using the new operator. This can be done as follows:

pointer_var = new data_type(value);

For eg:

int *p = new int(25); //allocates the memory for int and initialises it to the value 25.
float *q = new float(7.5);

DYNAMIC MEMORY ALLOCATION WITH NEW AND DELETE

Example:

```
#include<iostream>
#include<string.h>
using namespace std;
int main()
{
 char *str = “Kantipur Engineering College”;
 int len = strlen(str);                  //get length of str
 char *ptr;                              // make a pointer that points to char
 ptr = new char[len+1];                  // set aside memory: string + ‘\0’
 strcpy(ptr,str);                        // copy str to new memory area ptr
 cout<<endl<<”ptr =”<<ptr;               // show that str is now in ptr
 delete ptr;
 return 0;
 }
```

OUTPUT:

```
ptr = kantipur engineering college
```

CONDITION AND LOOPING

1) While Loop

It is the simplest kind of loop in which the condition is checked before entering the loop. Its syntax is:

```
while(condition)
{
  //statements
}
```

The while-loop simply repeats the statement while the condition is true. If the condition no longer holds true, the loop ends, and the program continues right after the loop.

While Loop Example

```
//custom countdown using while loop
#include<iostream>
Using namespace std;
int main()
{
       int n = 10;
       while(n>0)
       {
              cout << n << ” , “ ;
              n --;
        }
        cout<<”Liftoff!”<<endl;
        return 0;
}
```

Output:

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, Liftoff!
```

CONDITION AND LOOPING

2) Do-While Loop

It is very similar to the while loop, except that the condition is evaluated after the execution of the statement i.e at the exit point of the loop, guaranteeing at least one time execution of the statement, even if the condition is never is never fulfilled.

```
do
{
  //statements
} (condition);
```

Do-While Loop Example

```
//echo machine
#include<iostream>
#include<string.h>
using namespace std;
int main()
{
    string str;
    do
    {
        cout<<”Enter text : “ ;
        getline(cin,str);
        cout<<”You entered : “<<str <<endl;
    } while( str != “Bye-Bye”);
    return 0;
}
```

Output:

```
Enter text: hello
You entered : hello
Enter text : what’s up?
You entered : what’s up?
Enter text: Bye-Bye
You entered : Bye-Bye
```

CONDITION AND LOOPING

3) For Loop

The for loop is designed to iterate a certain number of times.

```
for( initialization ; condition ; update)
{
        //body of the loop
}
```

Like the while loop, this loop repeats the statement while the condition is true. But, in addition, the for loop provides specific locations to contain an initialization, condition and an increase or decrease expression. Therefore, it is especially useful to use counter variables as conditions.

CONDITION AND LOOPING

For Loop

It works in the following way:

- Initialization is executed at first. It initializes the variable to an initial value. This is executed single time, at the beginning of the loop.
- Condition is checked. If it is true, the loop continues, otherwise, the loop ends and the statement is skipped.
- If the condition is true, then the statement is executed. As usual, it can either be a single statement or a block of statements enclosed in curly braces { }.
- Update is executed and the loop gets back to step 2.

For Loop Example

```
//countdown using a for loop
#include<iostream>
using namespace std;
int main()
{
    for( int n = 10 ; n > 0 ; n–)
    {
        cout << n << ” , “ ;
    }
    cout<<”liftoff!”<<endl;
    return 0;
}
```

Output:

```
10, 9, 8, 7. 6, 5, 4, 3, 2, 1, Liftoff!
```

FUNCTIONS

Function Syntax

Function Declaration/Prototype:

```
return_type function_name(parameter list);
```

Function Definition:

```
return_type function_name(parameter list)
{
body_of _the_function
}
```

Function Call:

```
function_name(list of variables or values);
```

FUNCTIONS

Function Overloading

- C++ enables several functions of the same name to be defined as long as these functions have different sets of parameters. This capability is called function overloading/function polymorphism.
- The correct function to be invoked is determined by checking the number, types and order of the arguments in the call.

Function Overloading Example

```
#include <iostream>
using namespace std;
void sum(int a, int b)
{
    cout<<"Function 1: The sum is : " << (a+b) << endl;
}
void sum(int a, double b, int c)
{
    cout<<"Function 2: The sum is : " << double(a+b+c) << endl;
}
int main()
{
    sum(20, 30);
    sum(20, 30.10, 40);
    return 0;
}
```

OUTPUT:

```
Function 1 : The sum is : 50
Function 2 : The sum is : 90.1
```

FUNCTIONS

Inline Functions

- Inline Function is a function which expands to a line where it is invoked or called instead of jumping to the function itself. It is declared using the keyword ***inline*** in the function definition.

```
inline return_type function_name(list of arguments)
{
        //body of the function
}
```

- Inline functions increase the speed of execution of the program but it also increases the size of the program. So it is used only for small functions.
- A function cannot be made inline:
  - If static variable is declared inside the function
  - If a function returns a value where return type is specified
  - If function contains loop, switch or goto
  - If a function is recursive

FUNCTIONS

Inline Functions

- The inline keyword is just a request to the compiler to make a function inline function. The compiler may ignore the request if the function definition is too long or complicated and compile the function as a normal function.
- All the inline functions must be defined before they are called.

Note: Inline functions serve the same function as #define macro (generally used in C) but it provides better type checking and do not require special care for parentheses.

FUNCTIONS

Default Arguments

Default arguments are the default values provided to the arguments of the function. The default values are assigned during function declaration. The default value is used if the value is not passed during the function call. Otherwise, it uses the passed value.

Default arguments must be provided from the rightmost parameter in the argument list.

```
float interest(float p; int time; int rate = 0.5);
float interest(float p = 1000; int time = 1; int rate);   //error
float interest(float p; int time = 1; int rate);     //error
```

In the function call, any argument in a function cannot have a default value unless all arguments appearing on its right have their default values.

Default Arguments Example

```
#include <iostream>
using namespace std;
float interest(float p = 1000, int time = 1, int rate = 0.5)
{
 return (p * time * rate);
}
int main()
{
        cout<< “Interest =” << interest() <<endl;
        cout << “Interest =” << interest(1200) << endl;
        cout << “Interest =” << interest(1200, 10) << endl;
        cout << “Interest =” << interest(1500, 10, 0.7) << endl;
        return 0;
}
```

A function having N default arguments can be invoked in N+1 different ways as shown by the above example.

FUNCTIONS

Default Arguments

Note: Ambiguity arising when both function overloading and function with default arguments are used must be avoided.

Eg:

```
void function(int x, int y=0);
void function(int x);
```

Ambiguity arises when the function is called using only one integer as argument.

FUNCTIONS

Pass by Reference

Usually when a function is called, the arguments are copied into function and the function works on the copied value. Thus, the original value is not changed. If the original value must be changed then arguments must be passed by reference. It can be achieved in two ways:

- using reference variable
- using pointer

<u>Passing by reference using reference variable vs using pointer</u>

- using a reference variable saves memory as memory is not allocated for a pointer variable.
- using a reference variable is easier as no referencing / dereferencing is required like in pointers.
- reference variable references only a particular address it was first used to reference and hence works like a constant pointer. Using a pointer allows us to use the same pointer to point different addresses when required.

Pass By Reference Example

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

//pass by value
void swap1(int a, int b)
{
 int temp = b;
 b = a;
 a = temp;
}

//pass by reference using reference variable
void swap2(int &a, int &b)
{
 int temp = b;
 b = a;
 a = temp;
}
//pass by reference using pointer
void swap3(int *a, int *b){
 int temp = *b;
 *b = *a;
 *a = temp;
}
int main()
{
 int a = 10, b = 20;
 swap1(a,b);
 cout << “a = ” << a << “, ” << “b = ” << b << endl;
 swap2(a,b);
 cout << “a = ” << a << “, ” << “b = ” << b << endl;
 swap3(&a,&b);
 cout << “a = ” << a << “, ” << “b = ” << b << endl;
 return 0;
}
```

Pass By Reference Example

OUTPUT

```
a = 10, b = 20
a = 20, b = 10
a = 10, b = 20
```

Note: only variables can be passed by reference. We cannot pass constants like swap(2,3). Compiler generates error message if constants are passed.

In the above example,

- swap1() uses pass by value, i.e., it copies the values to the arguments of function. So, ‘a’ and ‘b’ in the main() function and ‘a’ and ‘b’ inside swap1() are different variables. Thus, in main() ‘a’ remains 10 and ‘b’ remains 20.
- swap2() uses pass by reference using a reference variable. So swap2() uses alias names to reference the variables in main(). ‘a’ is referenced as ‘a’ and ‘b’ is referenced as ‘b’ (they can be referenced by different names of course). When ‘a’ and ‘b’ are swapped in swap2(),the change is reflected in main() too.
- swap3() uses pass by reference using pointers. swap3() uses pointers to directly access the address of the variables in main() and, hence, directly swaps the values stored in the two memory locations.

FUNCTIONS

Return By Reference

- In C++, a function can return a variable by reference.
- Return by reference means a function is returning an alias of the variable in the return statement so the variable which is being returned should have a scope where the function is being invoked.
- Normally, Global variables and a variable which is being passed by reference to the function have a scope where the function is being invoked.
- Return by reference allows value to be assigned to the variable returned i.e. the function call can be used on the left side of the assignment operator to assign any value.

Note: Scope of the variables should be carefully chosen. A local variable of a function cannot be returned by referenced and generates an error. This is because local variables has a scope within the function only and is destroyed when function execution is completed. Thus, returning by reference will try to reference a variable that does not exist causing the error.

Return By Reference Example

```
#include <iostream>
using namespace std;
int &max(int&, int&);
int main()
{
 int a, b;
 cout << "Enter the numbers: " << endl;
 cin >> a >> b;
 cout << "The values are " << a << " and " << b
<< endl;
 cout << "The maximum value is " << max(a,b)
<< endl;
 max(a, b) = -1;
 cout << "The values are " << a << " and " << b
<< endl;
 return 0;
}
```

```
int &max(int& x, int& y){
 return (x>y?x:y);
}
```

OUTPUT:

```
Enter the numbers:
34
45
The values are 34 and 45
The maximum value is 45
The values are 34 and -1
```

ARRAY, POINTER AND STRING

Arrays

Array is the collection of similar data items that can be represented by a single variable name. The individual data items in an array are called elements. The data items are stored in contiguous memory locations.

data_type array_name[n]; *//allocates memory for the n different elements*

Eg:

int a[10];

Pointers

A pointer is a special type of variable which holds the address or location of another variable (or data item). Using a pointer, the data item is accessed indirectly through its address. In C++, arrays and pointers are closely related. An array name is treated as a pointer constant, and pointers can be subscripted like arrays.

data_type *pointer_name = &variable; *//pointer points to the variable*

ARRAY, POINTER AND STRING

Strings

String is an array of characters. However, string is treated separately than a general array, because they are used in some specific purpose like text manipulation.

char string_name[size];

THANK YOU