PROGRAMMING IN – C++

Input and Output:

For input and output we use the preprocessor directive **#inlude<iostream>**

Example:

```
#include<iostream>

using namespace std;

int main() {

   int a;

   cin>>a;

   cout<<a<<endl;

}
```

Array and Strings:

- **FIXED SIZE ARRAY:**

```
// Array_Fixed_Size_c++.cpp
#include <iostream>
int main() {
        short age[4];
        age[0] = 23;
        age[1] = 34;
        age[2] = 65;
        age[3] = 74;
        std::cout << age[0] << " ";
        std::cout << age[1] << " ";
        std::cout << age[2] << " ";
        std::cout << age[3] << " ";
        return 0;
}
```

- **FIXED SIZE VECTOR:**

```
// Array_Macro_c++.cpp
#include <iostream>
#include <vector>
using namespace std;
#define MAX 100
int main() {
        vector<int> arr(MAX); // Define-time size
        cout <<"Enter the no. of elements: ";
        int count, sum = 0;
        cin >>count;
```

```
        for(int i = 0; i < count; i++) {
                arr[i] = i; sum + = arr[i];
        }
        cout << "Array Sum: " << sum << endl;
}
```

- **DYNAMICALLY MANAGED ARRAY:**

```
// Array_Resize_c++.cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
        cout << "Enter the no. of elements: ";
        int count, sum=0;
        cin >> count;
        vector<int> arr; // Default size
        arr.resize(count); // Set resize
        for(int i = 0; i < arr.size(); i++) {
                arr[i] = i; sum + = arr[i];

        }

        cout << "Array Sum: " << sum << endl;

}
```

- **CONCATINATE STRINGS IN C AND CPP:**

---------- In C ----------

```
// Add_strings.c
#include <stdio.h>
#include <string.h>
int main() {
        char str1[] = {'H','E','L','L','O',' ','\0'};
        char str2[] = "WORLD";
        char str[20];
        strcpy(str, str1);
        strcat(str, str2);
        printf("%s\n", str);
}
```

---------- In CPP ----------

```
// Add_strings_c++.cpp
#include <iostream>
#include <string>
using namespace std;
int main(void) {
        string str1 = "HELLO ";
        string str2 = "WORLD";
        string str = str1 + str2;
        cout << str;
}
```

- **OTHER LIBRARYES ON STRINGS:**

Function	Description	Used Frequently?
Copying: `memcpy`	Copy block of memory (function)	Yes
`memmove`	Move block of memory (function)	Yes
`strcpy`	Copy string (function)	Yes
`strncpy`	Copy characters from string (function)	
Concatenation: `strcat`	Concatenate strings (function)	Yes
`strncat`	Append characters from string (function)	
Comparison: `memcmp`	Compare two blocks of memory (function)	
`strcmp`	Compare two strings (function)	Yes
`strcoll`	Compare two strings using locale (function)	
`strncmp`	Compare characters of two strings (function)	
`strxfrm`	Transform string using locale (function)	
Searching: `memchr`	Locate character in block of memory (function)	Yes
`strchr`	Locate first occurrence of character in string (function)	Yes
`strcspn`	Get span until character in string (function)	
`strpbrk`	Locate characters in string (function)	
`strrchr`	Locate last occurrence of character in string (function)	
`strspn`	Get span of character set in string (function)	
`strstr`	Locate substring (function)	Yes
`strtok`	Split string into tokens (function)	Yes
Other: `memset`	Fill block of memory (function)	
`strerror`	Get pointer to error message string (function)	
`strlen`	Get string length (function)	Yes
Macros: `NULL`	Null pointer (macro)	Yes
Types: `size_t`	Unsigned integral type (type)	Yes

Function	Description (Member Function)	C Parallel
Member functions		
`(constructor)`	Construct string object (public)	Initialize `string` object with a C string
`(destructor)`	String destructor (public)	
`operator=`	String assignment (public)	`strcpy()`. `operator=` does shallow copy
Iterators		Iteration done explicitly by loop index
`begin`	Return iterator to beginning (public)	
`end`	Return iterator to end (public)	
`rbegin`	Return reverse iterator to reverse beginning (public)	
`rend`	Return reverse iterator to reverse end (public)	
`cbegin`	Return const_iterator to beginning (public)	
`cend`	Return const_iterator to end (public)	
`crbegin`	Return const_reverse_iterator to reverse beginning (public)	
`crend`	Return const_reverse_iterator to reverse end (public)	

Function	Description (Member Function)	C Parallel
Capacity		
`size`	Return length of string (public)	`strlen()`
`length`	Return length of string (public)	`strlen()`
`max_size`	Return maximum size of string (public)	Fixed at allocation
`resize`	Resize string (public)	
`capacity`	Return size of allocated storage (public)	Need to be remembered in the code
`reserve`	Request a change in capacity (public)	
`clear`	Clear string (public)	`strcpy()` an empty string
`empty`	Test if string is empty (public)	`strlen() == 0`
`shrink_to_fit`	Shrink to fit (public)	
String operations		
`c_str`	Get C string equivalent (public)	C string from a `string` object
`data`	Get string data (public)	
`get_allocator`	Get allocator (public)	
`copy`	Copy sequence of characters from string (public)	`strncpy()`
`find`	Find content in string (public)	`strchr()`, `strstr()`
`rfind`	Find last occurrence of content in string (public)	
`find_first_of`	Find character in string (public)	`strchr()`
`find_last_of`	Find character in string from the end (public)	`strrchr()`
`find_first_not_of`	Find absence of character in string (public)	
`find_last_not_of`	Find non-matching character in string from the end (public)	
`substr`	Generate substring (public)	`strncpy()`
`compare`	Compare strings (public)	`strcmp()`

Function	Description (Member Function)	C Parallel
Element access		
operator[]	Get character of string (public)	operator[]
at	Get character in string (public)	operator[]
back	Access last character (public)	Character at strlen()-1
front	Access first character (public)	Character at 0^{th} location
Modifiers		
operator+=	Append to string (public)	strcat()
append	Append to string (public)	strcat()
push_back	Append character to string (public)	Set character to strlen() and NULL to next location
assign	Assign content to string (public)	
insert	Insert into string (public)	
erase	Erase characters from string (public)	
replace	Replace portion of string (public)	
swap	Swap string values (public)	Character by character swapping between two arrays
pop_back	Delete last character (public)	Set location strlen()-1 to NULL
Member constants		
npos	Maximum value for size_t (public static)	
Non-member function overloads		
operator+	Concatenate strings (global)	strcat()
relational operators	Relational operators for string (global)	strcmp() followed by tests for -1, 0, +1
swap	Exchanges the values of two strings (global)	
operator>>	Extract string from stream (global)	format %s
operator<<	Insert string into stream (global)	format %s
getline	Get line from stream into string (global)	getline() in <stdlib.h>

Sorting and Searching:

- **BUBBLE SORT:**

```
#include <iostream>
using namespace std;
int main() {
        int data[] = {32, 71, 12, 45, 26};
        int n = 5, temp;
        for(int step = 0; step < n - 1; ++step){
                for(int i = 0;i < n-step-1; ++i) {
                        if (data[i] > data[i+1]) {
                                temp = data[i];
                                data[i] = data[i+1];
                                data[i+1] = temp;

                        }

                }

        }
                for(int i = 0; i < n; ++i)

                cout << data[i] << " ";

}
```


- **USING SORT FROM STD LIBRARY:**

----------In C----------

```
#include <stdio.h>
#include <stdlib.h> // qsort function
// compare Function Pointer
int compare(const void *a, const void *b) { // Type unsafe
        return (*(int*)a < *(int*)b); // Cast needed
```

```
}
int main () {
        int data[] = {32, 71, 12, 45, 26};
        // Start ptr., # elements, size, func. ptr.
        qsort(data, 5, sizeof(int), compare);
        for(int i = 0; i < 5; i++)
                printf ("%d ", data[i]);
}
```

----------In CPP----------

```
#include <iostream>
#include <algorithm> // sort function
using namespace std;
// compare Function Pointer
bool compare(int i, int j) { // Type safe
        return (i > j); // No cast needed
}
int main() {
        int data[] = {32, 71, 12, 45, 26};
        // Start ptr., end ptr., func. ptr.
        sort(data, data+5, compare);
        for (int i = 0; i < 5; i++)
                cout << data[i] << " ";
}
```

NOTE: In default sort no need to pass function name and the output will be in ascending order i.e. increasing order

- **SEARCHING IN CPP:**

----------In C----------

```
#include <stdio.h>
#include <stdlib.h> // bsearch function
// compare Function Pointer
int compare(const void * a, const void * b) { // Type unsafe
        if (*(int*)a<*(int*)b) return -1; // Cast needed
        if (*(int*)a==*(int*)b) return 0; // Cast needed
        if (*(int*)a>*(int*)b) return 1; // Cast needed
}
int main () {
        int data[] = {1,2,3,4,5}, k = 3;
        if (bsearch(&k, data, 5, sizeof(int), compare))
        printf("found!\n");
        else printf("not found\n");
}
```

----------In CPP----------

```
#include <iostream>
#include <algorithm> // binary_search function
using namespace std;

int main() {
        int data[] = {1,2,3,4,5}, k = 3;
```

```
        if (binary_search(data, data+5, k))
                cout << "found!\n";
        else cout << "not found\n";
}
```

- **ALGORITHM LIBRARY:**

```
----------Replace----------
// Replace.cpp
#include <iostream>
#include <algorithm> // replace function
using namespace std;
int main() {
        int data[] = {1, 2, 3, 4, 5};
        replace(data, data+5, 3, 2);
        for(int i = 0; i < 5; ++i)
                cout << data[i] << " ";
        return 0;
}
```

Output:
1 2 2 4 5

```
----------Rotate----------
// Rotate.cpp
#include <iostream>
#include <algorithm> // rotate function
using namespace std;
int main() {
        int data[] = {1, 2, 3, 4, 5};
        rotate(data, data+2, data+5);
        for(int i = 0; i < 5; ++i)
                cout << data[i] << " ";
        return 0;

}
```

Output:
3 4 5 1 2

- **Other functions**:
  replace, merge, swap, remove

Stack and Common Data Structures:

- **STACK IN C:**

```
----------Reversing String----------
#include <stdio.h>
typedef struct stack {
        char data [100];
        int top;
} stack;
int empty(stack *p) {
        return (p->top == -1);
```

```
}
int top(stack *p) {
        return p -> data [p->top];
}
void push(stack *p, char x) {
        p -> data [++(p -> top)] = x;
}
void pop(stack *p) {
        if (!empty(p)) (p->top) = (p->top) -1;
}
int main() {
        stack s;
        s.top = -1;
        char ch, str[10] = "ABCDE";
        int i, len = sizeof(str);
        for(i = 0; i < len; i++)
                push(&s, str[i]);
        printf("Reversed String: ");
        while (!empty(&s)) {
                printf("%c ", top(&s));
                pop(&s);
        }
}
```

----------Postfix Expression Evaluation----------

```
#include <stdio.h>
typedef struct stack {
        char data [100];
        int top;
} stack;
int empty(stack *p) {
        return (p->top == -1);
}
int top(stack *p) {
        return p -> data [p->top];
}
void push(stack *p, char x) {
        p -> data [++(p -> top)] = x;
}
void pop(stack *p) {
        if (!empty(p)) (p->top) = (p->top) -1;
}
void main() {
        stack s; s.top = -1;
        // Postfix expression: 1 2 3 * + 4 -
        char postfix[] = {'1','2','3','*','+','4','-'};
        for(int i = 0; i < 7; i++) {
```

```
                char ch = postfix[i];
                if (isdigit(ch)) push(&s, ch-'0');
                else {
                        int op2 = top(&s); pop(&s);
                        int op1 = top(&s); pop(&s);
                        switch (ch) {
                                case '+': push(&s, op1 + op2); break;
                                case '-': push(&s, op1 - op2); break;
                                case '*': push(&s, op1 * op2); break;
                                case '/': push(&s, op1 / op2); break;
                        }
                }
        }
        printf("Evaluation %d\n", top(&s));
}
```

- **STACK IN CPP:**

```
----------Reverse a String in C++----------
#include <stdio.h>
#include <string.h>
#include "stack.h" // User defined codes
int main() {
        char str[10] = "ABCDE";
        stack s; s.top = -1; // stack struct
        for(int i = 0; i < strlen(str); i++)
                push(&s, str[i]);
        printf("Reversed String: ");
        while (!empty(&s)) {
                printf("%c ", top(&s)); pop(&s);
        }
}
#include <iostream>
#include <cstring>
#include <stack> // Library codes
using namespace std;
int main() {
        char str[10]= "ABCDE";
        stack<char> s; // stack class
        for(int i = 0; i < strlen(str); i++)
                s.push(str[i]);
        cout << "Reversed String: ";
        while (!s.empty()) {
                cout << s.top(); s.pop();
        }
}
----------Postfix Evaluation----------
#include <iostream>
#include <stack> // Library codes
using namespace std;
```

```
int main()
{
        // Postfix expression: 1 2 3 * + 4 -
        char postfix[] ={'1','2','3','*','+','4','-'}, ch;
        stack<int> s; // stack class
        for(int i = 0; i < 7; i++) {
                ch = postfix[i];
                if (isdigit(ch)){
                        s.push(ch-'0');
                }
                else {
                        int op1 = s.top();
                        s.pop();
                        int op2 = s.top();
                        s.pop();
                        switch (ch) {
                                case '*': s.push(op2 * op1); break;
                                case '/': s.push(op2 / op1); break;
                                case '+': s.push(op2 + op1); break;
                                case '-': s.push(op2 - op1); break;

                        }

                }

        }
        cout << "\nEvaluation " << s.top();
}
```

- **Other Containers in CPP:**

Container	Class Template	Remarks
Sequence containers: *Elements are ordered in a strict sequence and are accessed by their position in the sequence*		
array (C++11)	Array class	1D array of *fixed-size*
vector	Vector	1D array of *fixed-size* that can *change in size*
deque	Double ended queue	*Dynamically sized*, can be expanded / contracted on *both ends*
forward_list (C++11)	Forward list	*Const. time insert / erase* anywhere, done as *singly-linked lists*
list	List	*Const. time insert / erase* anywhere, iteration in *both directions*
Container adaptors: *Sequence containers adapted with specific protocols of access like LIFO, FIFO, Priority*		
stack	LIFO stack	Underlying container is deque (default) or as specified
queue	FIFO queue	Underlying container is deque (default) or as specified
priority_queue	Priority queue	Underlying container is vector (default) or as specified
Associative containers: *Elements are referenced by their key and not by their absolute position in the container. They are typically implemented as binary search trees and needs the elements to be comparable*		
set	Set	Stores *unique elements* in *a specific order*
multiset	Multiple-key set	Stores elements in *an order* with *multiple equivalent values*
map	Map	Stores *<key, value>* in *an order* with *unique keys*
multimap	Multiple-key map	Stores *<key, value>* in *an order* with *multiple equivalent values*
Unordered associative containers: *Elements are referenced by their key and not by their absolute position in the container. Implemented using a hash table of keys and has fast retrieval of elements based on keys*		
unordered_set (C++11)	Unordered Set	Stores *unique elements* in no particular order
unordered_multiset (C++11)	Unordered Multiset	Stores elements in *no order* with *multiple equivalent values*
unordered_map (C++11)	Unordered Map	Stores *<key, value>* in *no order* with *unique keys*
unordered_multimap (C++11)	Unordered Multimap	Stores *<key, value>* in *no order* with *multiple equivalent values*

Programming in Modern C++ Partha Pratim Das M05.16

Constants and Inline Functions:

- **POINTER TO CONSTANT DATA:**

```
int m = 4;
const int n = 5;
const int * p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
p = &m; // Okay
*p = 8; // Error: p points to a constant data. Its pointee cannot be changed
```

Interestingly,

```
int n = 5;
const int * p = &n;
...
n = 6; // Okay
*p = 6; // Error: p points to a constant data (n) that cannot be changed
```

Finally,

```
const int n = 5;
int *p = &n; // Error: If this were allowed, we would be able to change constant n
...
n = 6; // Error: n is constant and cannot be changed
*p = 6; // Would have been okay, if declaration of p were valid
```

- **CONSTANT POINTER:**

```
int m = 4, n = 5;
int * const p = &n;
...
n = 6; // Okay
*p = 7; // Okay. Both n and *p are 7 now
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

By extension, both can be const

```
const int m = 4;
const int n = 5;
const int * const p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

Finally, to decide on const-ness, draw a mental line through *

```
int n = 5;
int * p = &n; // non-const-Pointer to non-const-Pointee
const int * p = &n; // non-const-Pointer to const-Pointee
int * const p = &n; // const-Pointer to non-const-Pointee
const int * const p = &n; // const-Pointer to const-Pointee
```

- **USING VOLATILE:**

```
static int i;
void fun(void) {
        i = 0;
```

```
        while (i != 100);
}
//This is an infinite loop! Hence the compiler should optimize as:
static int i;
void fun(void) {
        i = 0;
        while (1); // Compiler optimizes
}
```

Now qualify i as volatile:

```
static volatile int i;
void fun(void) {
        i = 0;
        while (i != 100); // Compiler does not optimize
}
```

Being volatile, i can be changed by hardware anytime. It waits till the value becomes 100 (possibly some hardware writes to a port).

- **INLINE FUNCTIONS:** Functions with one parameter.
  Examples:

  1.

```
#include <iostream>
using namespace std;
#define SQUARE(x) x * x
int main() {
        int a = 3, b;
        b = SQUARE(a);
        cout << "Square = " << b << endl;
}
```

  Output:
  Square = 9

  2.

```
#include <iostream>
using namespace std;
#define SQUARE(x) x * x
int main() {
        int a = 3, b;
        b = SQUARE(a + 1); // Error: Wrong macro expansion
        cout << "Square = " << b << endl;
}
```

  Output:
  Square = 7

  3.

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

```
int a = 3, b;
b = SQUARE(++a);
cout << "Square = " << b << endl;
}
```

Output:

Square = 16

4.

```
#include <iostream>
using namespace std;
inline int SQUARE(int x) { return x * x; }
int main() {
int a = 3, b;
b = SQUARE(a);
cout << "Square = " << b << endl;
}
```

Output:

Square = 9

Note: Here SQUARE(a + 1) works • SQUARE(++a) works • SQUARE(++a) checks type

Reference and Pointer:

- **RFERANCE VARIABLES:**

  A reference is an alias / synonym for an existing variable

```
int i = 15; // i is a variable
int &j = i; // j is a reference to i
```

Example:

```
#include <iostream>
using namespace std;
int main() {
        int a = 10, &b = a; // b is reference of a
        // a and b have the same memory location
        cout << "a = " << a << ", b = " << b << ". " << "&a = " << &a << ", &b = " << &b << endl;
        ++a; // Changing a appears as change in b
        cout << "a = " << a << ", b = " << b << endl;
        ++b; // Changing b also changes a
        cout << "a = " << a << ", b = " << b << endl;
}
```

Output:

a = 10, b = 10. &a = 002BF944, &b = 002BF944

a = 11, b = 11

a = 12, b = 12

Note: A reference can't be NULL, similarly a constant can't be NULL. A reference must be constant in order to refer a constant.

Example:

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

```
        int i = 2;
        int& j = i;
        const int& k = 5; // const tells compiler to allocate a memory with the value 5
        const int& l = j + k; // Similarly for j + k = 7 for l to refer to
        cout << i << ", " << &i << endl; // Prints: 2, 0x61fef8
        cout << j << ", " << &j << endl; // Prints: 2, 0x61fef8
        cout << k << ", " << &k << endl; // Prints: 5, 0x61fefc
        cout << l << ", " << &l << endl; // Prints: 7, 0x61ff00
}
```

- **CALL BY REFERANCE:**

```
#include <iostream>
using namespace std;
void Function_under_param_test (int&, /*Reference parameter*/,int); // Value parameter
int main() {
        int a = 20;
        cout << "a = " << a << ", &a = " << &a << endl << endl;
        Function_under_param_test(a, a); // Function call
}
void Function_under_param_test(int &b, int c) { // Function definition
        cout << "b = " << b << ", &b = " << &b << endl << endl;
        cout << "c = " << c << ", &c = " << &c << endl << endl;
}
```

------- Output -------

a = 20, &a = 0023FA30
b = 20, &b = 0023FA30 // Address of b is same as a as b is a reference of a
c = 20, &c = 0023F95C // Address different from a as c is a copy of a

- **RETURN BY REFARENCE:**

```
#include <iostream>
using namespace std;
int& Function_Return_By_Ref(int &x) {
        cout << "x = " << x << " &x = " << &x << endl;
        return (x);
}
int main() {
        int a = 10;
        cout << "a = " << a << " &a = " << &a << endl;
        const int& b = // const optional
        Function_Return_By_Ref(a);
        cout << "b = " << b << " &b = " << &b << endl;
}
```

Output:

a = 10 &a = 00A7F8FC
x = 10 &x = 00A7F8FC
b = 10 &b = 00A7F8FC // Reference to a

- **RETURN BY VALUE:**

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

```
int Function_Return_By_Val(int &x) {
        cout << "x = " << x << " &x = " << &x << endl;
        return (x);
}
int main() {
        int a = 10;
        cout << "a = " << a << " &a = " << &a << endl;
        const int& b = // const needed. Why?
        Function_Return_By_Val(a);
        cout << "b = " << b << " &b = " << &b << endl;
}
```

Output:

a = 10 &a = 00DCFD18
x = 10 &x = 00DCFD18
b = 10 &b = 00DCFD00 // Reference to temporary,

DEFAULT PERAMETERS AND FUNCTION OVERLOADING:

- **DEFAULT PERAMETER:**

```
#include <iostream>
using namespace std;
int IdentityFunction(int a = 10) { // Default value for parameter a
        return (a);
}
int main() {
        int x = 5, y;
        y = IdentityFunction(x);
        cout << "y = " << y << endl;
        y = IdentityFunction();
        cout << "y = " << y << endl;
}
```

Output:

y = 5
y = 10

- **IMPORTANT POINTS:**

• All parameters to the right of a parameter with default argument must have default arguments (function f violates)
• Default arguments cannot be re-defined (second signature of function g violates)
• All non-defaulted parameters needed in a call (first call of g() violates)

Example:

```
#include <iostream>
void f(int, double = 0.0, char *);
// Error C2548: f: missing default parameter for parameter 3
void g(int, double = 0, char * = NULL); // OK
void g(int, double = 1, char * = NULL);
// Error C2572: g: redefinition of default parameter : parameter 3
// Error C2572: g: redefinition of default parameter : parameter 2
int main() {
```

```
int i = 5; double d = 1.2; char c = 'b';
g(); // Error C2660: g: function does not take 0 arguments
g(i);
g(i, d);
g(i, d, &c);
}
```

- **FUNCTION OVERLOADING RULES:**
  Two functions having the same signature but different return types cannot be overloaded
  • The same function name may be used in several definitions
  • Functions with the same name must have different number of formal parameters and/or different types of formal parameters
  • Function selection is based on the number and the types of the actual parameters at the places of invocation
  • Function selection (Overload Resolution) is performed by the compiler
  • Two functions having the same signature but different return types will result in a compilation error due to attempt to re-declare
  • Overloading allows Static Polymorphism
- **OVERLOAD RESOLUTION:**
  To resolve overloaded functions with one parameter
  ◦ Identify the set of Candidate Functions
  ◦ From the set of candidate functions identify the set of Viable Functions
  ◦ Select the Best viable function through (Order is important)
  . Exact Match
  . Promotion
  . Standard type conversion
  . User defined type conversion
  Example: a)
  In the context of a list of function prototypes:

```
int g(double); // F1
void f(); // F2
void f(int); // F3
double h(void); // F4
int g(char, int); // F5
void f(double, double = 3.4); // F6
void h(int, double); // F7
void f(char, char *); // F8
```

  The call site to resolve is:

```
f(5.6);
```

  • Resolution:
  ◦ Candidate functions (by name): F2, F3, F6, F8
  ◦ Viable functions (by # of parameters): F3, F6
  ◦ Best viable function (by type double – Exact Match): F6
  b)

```
#include <iostream>
using namespace std;
// Overloaded Area functions
int Area(int a, int b = 10) { return (a * b); }
double Area(double c, double d) { return (c * d); }
```

```
int main() {
        int x = 10, y = 12, t; double z = 20.5, u = 5.0, f;
        t = Area(x); // Binds int Area(int, int = 10)
        cout << "Area = " << t << endl; // Area = 100
        t = Area(x, y); // Binds int Area(int, int = 10)
        cout << "Area = " << t << endl; // Area = 120
        f = Area(z, u); // Binds double Area(double, double)
        cout << "Area = " << f << endl; // Area = 102.5
        f = Area(z); // Binds int Area(int, int = 10)
        cout << "Area = " << f << endl; // Area = 200
        // Un-resolvable between int Area(int a, int b = 10) and double Area(double c,
double d)
        f = Area(z, y); // Error: call of overloaded Area(double&, int&) is ambiguous
}
```

c)

```
#include <iostream>
using namespace std;
int f();
int f(int = 0);
int f(int, int);
int main() {
        int x = 5, y = 6;
        f(); // Error C2668: f: ambiguous call to overloaded function
        // More than one instance of overloaded function f
        // matches the argument list:
        // function f()
        // function f(int = 0)
        f(x); // int f(int);
        f(x, y); // int f(int, int);
        return 0;
}
```


OPERATOR OVERLOADING:

- is a specific case of polymorphism, where different operators have different implementations depending on their arguments

Example: a)

```
#include <iostream>
#include <cstring>
using namespace std;
typedef struct _String { char *str; } String;
String operator+(const String& s1, const String& s2) {
        String s;
        s.str = (char *) malloc(strlen(s1.str) +
        strlen(s2.str) + 1); // Allocation
        strcpy(s.str, s1.str); strcat(s.str, s2.str);
        return s;
}
```

```
int main() {
        String fName, lName, name;
        fName.str = strdup("Partha ");
        lName.str = strdup("Das");
        name = fName + lName; // Overloaded operator +
        cout << "First Name: " << fName.str << endl;
        cout << "Last Name: " << lName.str << endl;
        cout << "Full Name: " << name.str << endl;
}
```

Output:
First Name: Partha
Last Name: Das
Full Name: Partha Das

- **RULES:**
  • No new operator such as operators** or operators<> can be defined for overloading
  • Intrinsic properties of the overloaded operator cannot be changed
  ◦ Preserves arity
  ◦ Preserves precedence
  ◦ Preserves associativity
  • These operators can be overloaded:
  [] + - * / % ^ & | ~ ! = += -= *= /= %= ^= &= |=
  << >> >>= <<= == != < > <= >= && || ++ -- , ->* -> ( ) [ ]
  • For unary prefix operators, use: MyType& operator++(MyType& s1)
  • For unary postfix operators, use: MyType operator++(MyType& s1, int)
  • The operators:: (scope resolution), operator. (member access), operator.* (member access through pointer to member), operator sizeof, and operator?: (ternary conditional) cannot be overloaded
  • The overloads of operators&&, operator||, and operator, (comma) lose their special properties: short-circuit evaluation and sequencing
  • The overload of operators-> must either return a raw pointer or return an object (by reference or by value), for which operators-> is in turn overloaded

DYNAMIC MEMORY MANAGEMENT:

- C++ introduces operators new and delete to dynamically allocate and de-allocate memory:
  Functions malloc() & free()----------
  ```
  #include <iostream>
  #include <cstdlib>
  using namespace std;
  int main() {
          int *p = (int *)malloc(sizeof(int));
          *p = 5;
          cout << *p; // Prints: 5
          free(p);
  }
  ```
  operator new & operator delete----------
  ```
  #include <iostream>
  using namespace std;
  ```

```
int main() {
        int *p = new int(5);
        cout << *p; // Prints: 5
        delete p;
}
```

- **FEW MORE EXAMPLES ON NEW AND DELETE:**

a)

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
        int *p = (int *)operator new(sizeof(int));
        *p = 5;
        cout << *p; // Prints: 5
        operator delete(p);
}
```

b)

```
#include <iostream>
using namespace std;
int main() {
        int *a = new int[3];
        a[0] = 10; a[1] = 20; a[2] = 30;
        for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "<< a[i] << " ";
        delete [] a;
}
```

Output:

a[0] = 10 a[1] = 20 a[2] = 30

- **RULES:**

Allocation and De-allocation must correctly match.

◦ Do not free the space created by new using free()

◦ And do not use delete if memory is allocated through malloc()

These may results in memory corruption

Allocator	De-allocator
malloc()	free()
operator new	operator delete
operator new[]	operator delete[]
operator new()	No delete

• Passing NULL pointer to delete operator is secure

• Prefer to use only new and delete in a C++ program

• The new operator allocates exact amount of memory from Heap or Free Store

• new returns the given pointer type – no need to typecast

• new, new[ ] and delete, delete[] have separate semantics

CLASSES AND OBJECTS:

- **Example: a)**

```
// File Name:Complex_object_c++.cpp
#include <iostream>
using namespace std;
class Complex {
        public: // class
                double re, im; // Data members
};
int main() {
        // Object c declared, initialized
        Complex c = { 4.2, 5.3 };
        cout << c.re << " " << c.im; // Use by dot
}
```

Output:

4.2 5.3

b)

```
// File Name:Rectangle_object_c++.cpp
#include <iostream>
using namespace std;
class Point {
        public: // class Point
                int x; int y; // Data members
};
class Rect {
        public: // Rect uses Point
                Point TL; // Top-Left. Member of UDT
                Point BR; // Bottom-Right. Member of UDT
};
int main() {
        Rect r = { { 0, 2 }, { 5, 7 } };
        // r.TL <-- { 0, 2 }; r.BR <-- { 5, 7 }
        // r.TL.x <-- 0; r.TL.y <-- 2
        // Rectangle Object r accessed
        cout << "[(" << r.TL.x << " " << r.TL.y <<") (" << r.BR.x << " " << r.BR.y << ")]";
}
```

Output:

[(0 2) (5 7)]

- **MEMBER FUNCTIONS:**

```
// File Name:Complex_func_c++.cpp
#include <iostream>
#include <cmath>
using namespace std;
// Type as UDT
class Complex {
        public: double re, im;
                double norm() {
                        return sqrt(re*re + im*im); }
                void print() {
```

```
                cout << "|"<< re<< "+j"<< im<< "| = ";
                cout << norm(); // Call method
        }
};
int main() {
        Complex c = { 4.2, 5.3 };
        c.print(); // Invoke method print of c
}
```

Output:

|4.2+j5.3| = 6.7624

- ***this* POINTER:**

  • An implicit this pointer holds the address of an object
  • this pointer serves as the identity of the object in C++
  • Type of this pointer for a class X object: X * const this;
  • this pointer is accessible only in member functions

  Example:

```
#include <iostream>
using namespace std;
class X {
        public: int m1, m2;
                void f(int k1, int k2) {
                        m1 = k1;
                        this->m2 = k2; // Explicit access with this pointer
                        cout << "Id = " << this << endl; // Identity (address) of the object
                }
};
int main() {
        X a;
        a.f(2, 3);
        cout << "Addr = " << &a << endl; // Address (identity) of the object
        cout << "a.m1 = " << a.m1 << " a.m2 = " << a.m2 << endl;
        return 0;
}
```

Output:

Id = 0024F918
Addr = 0024F918
a.m1 = 2 a.m2 = 3

ACCESS SPECIFIERS:

- private — accessible inside the definition of the class
  . member functions of the same class
  ◦ public — accessible everywhere
  . member functions of the same class
  . member function of a different class
  . global functions

  Example:

```
#include <iostream>
```

```
#include <cmath>
using namespace std;
class Complex {
        private: double re, im;
        public:
                double norm() { return sqrt(re*re + im*im); }
        };
void print(const Complex& t) {
        // Global fn.
        cout << t.re << "+j" << t.im << endl;
        // Complex::re / Complex::im: cannot access
}
int main() {
        Complex c = { 4.2, 5.3 }; // Error
        // 'initializing': cannot convert from
        // 'initializer-list' to 'Complex'
        print(c);
        cout << c.norm();
}
```

CONSTRUCTOR, DESTRUCTOR AND OBJECT LIFETIME:

- **CONSTRUCTOR:**

Examples: **a)**

```
#include <iostream>
using namespace std;
class Stack {
        private:
                char data_[10]; int top_;
        public: Stack() : top_(-1) { } // Initialization
                int empty() { return (top_ == -1); }
                void push(char x) { data_[++top_] = x; }
                void pop() { --top_; }
                char top() { return data_[top_]; }
};
int main() {
        char str[10] = "ABCDE";
        Stack s; // Init by Stack::Stack() call
        for (int i = 0; i < 5; ++i) s.push(str[i]);
        while(!s.empty()) { cout << s.top(); s.pop(); }
}
```

b) Dynamic Array

```
#include <iostream>
using namespace std;
class Stack {
        private:
                char *data_; int top_; // Dynamic
        public: Stack(); // Constructor
```

```
};
Stack::Stack(): data_(new char[10]), // Init List
top_(-1) { cout << "Stack::Stack()" << endl;
}
int main() {
        char str[10] = "ABCDE";
        Stack s; // Init by Stack::Stack() call
        for (int i=0; i<5; ++i) s.push(str[i]);
        while(!s.empty()) { cout << s.top(); s.pop(); }
}
```

Output:

Stack::Stack()

EDCBA

- **DESTRUCTOR:**

Example: **a)**

```
#include <iostream>
using namespace std;
class Stack {
        char *data_; int top_;
        public: Stack(): data_(new char[10]), top_(-1)
                { cout << "Stack() called\n"; }
                ~Stack() { cout << "\n~Stack() called\n";
                        delete [] data_;
                }
};
int main() { char str[10] = "ABCDE";
Stack s;
} // De-Init by automatic Stack::~Stack() call
```

Output:

Stack() called

EDCBA

~Stack() called

- **STORAGE CLASS SPECIFIERS:**

• The storage class specifiers are a part of the decl-specifier-seq of a name's declaration syntax

• Together with the scope of the name, they control two independent properties of the name: its

storage duration (or Lifetime) and its linkage

◦ auto or no specifier: Automatic storage duration. Deprecated in C++11 and used for a difference semantics (Module 46)

◦ register: Automatic storage duration that hints to the compiler to place the object in the processor's register. Deprecated in C++17

◦ static: Static (or thread storage as discussed in Module 59) duration and internal linkage (or external linkage for static class members not in an anonymous namespace)

◦ extern: Static (or thread storage as discussed in Module 59) duration and external linkage

◦ thread local: Thread storage duration in concurrency support since C++11 (Module 59)

◦ mutable: Related to const / volatile does not affect storage duration or linkage

- **CREATING OBJECT WITH new :**

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex {
        private: double re_, im_;
        public:
                Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
                { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
                ~Complex() { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; } // Dtor
                double norm() { return sqrt(re_*re_ + im_*im_); }
                void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
int main() {
        unsigned char buf[100]; // Buffer for placement of objects
        Complex* pc = new Complex(4.2, 5.3); // new: allocates memory, calls Ctor
        Complex* pd = new Complex[2]; // new []: allocates memory
        Complex* pe = new (buf) Complex(2.6, 3.9);
        pc->print();
        pd[0].print(); pd[1].print();
        pe->print();
        delete pc;
        delete [] pd;
        pe->~Complex();
}
```


COPY CONSTRUCTOR AND COPY ASSIGNMENT OPERATOR:

- **Order of Initialization:**

First declared is initialized first (First come first serve)

Example:

```
#include <iostream>
using namespace std;
int init_m1(int m) {
        // Func. to init m1_
        cout << "Init m1_: " << m << endl;
        return m;
}
int init_m2(int m) {
        // Func. to init m2_
        cout << "Init m2_: " << m << endl;
        return m;
}
class X {
        int m2_; // Order of data members swapped
        int m1_;
        public: X(int m1, int m2) :
                m1_(init_m1(m1)), // Called 2nd
```

```
                m2_(init_m2(m2)) // Called 1st
                { cout << "Ctor: " << endl; }
                ~X() { cout << "Dtor: " << endl; } };
int main() {

        X a(2, 3);
        return 0;

}
```

Output:
Init m2_: 3
Init m1_: 2
Ctor:
Dtor:

- **COPY CONSTRUCTOR:**

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex {
        double re_, im_;
        public:
                // Constructor
                Complex(double re, double im): re_(re), im_(im)
                { cout << "Complex ctor: "; print(); }
                // Copy Constructor
                Complex(const Complex& c): re_(c.re_), im_(c.im_)
                { cout << "Complex copy ctor: "; print(); }
                // Destructor
                ~Complex()
                { cout << "Complex dtor: "; print(); }
                double norm() { return sqrt(re_*re_ + im_*im_); }
                void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
int main() {
        Complex c1(4.2, 5.3), // Constructor - Complex(double, double)
        c2(c1), // Copy Constructor - Complex(const Complex&)
        c3 = c2; // Copy Constructor - Complex(const Complex&)
        c1.print(); c2.print(); c3.print();
}
```

- **SIGNATURE OF COPY CONSTRUCTOR:**

• Signature of a Copy Constructor can be one of:

```
MyClass(const MyClass& other); // Common

// Source cannot be changed

MyClass(MyClass& other); // Occasional

// Source needs to change. Like in smart pointers
```

MyClass(volatile const MyClass& other); // Rare
MyClass(volatile MyClass& other); // Rare
• None of the following are copy constructors, though they can copy:
MyClass(MyClass* other);
MyClass(const MyClass* other);
• Why the parameter to a copy constructor must be passed as Call-by-Reference?
MyClass(MyClass other);
The above is an infinite recursion of copy calls as the call to copy constructor itself needs to make copy for the Call-by-Value mechanism

- **SHALLOW COPY:**
  • Consider a class:

```
class A {
        int i_; // Non-pointer data member
        int* p_; // Pointer data member
        public:
                A(int i, int j) : i_(i), p_(new int(j)) { } // Init. with pointer to dynamically
        created object
                ~A() { cout << "Destruct " << this << ": "; // Object identity
                cout << "i_ = " << i_ << " p_ = " << p_ << " *p = " << *p_ << endl; // Object
        state
                delete p_; // Release resource
                }
};
```

  • As no copy constructor is provided, the implicit copy constructor does a bit-wise copy. So when an A object is copied, p is copied and continues to point to the same dynamic int:

```
int main() {
        A a1(2, 3); A a2(a1); // Construct a2 as a copy of a1. Done by bit-wise copy
        cout << "&a1 = " << &a1 << " &a2 = " << &a2 << endl;
}
```

  • The output is wrong, as a1.p = a2.p points to the same int location. Once a2 is destructed, a2.p is released, and a1.p becomes dangling. The program may print garbage or crash:
  &a1 = 008FF838 &a2 = 008FF828 // Identities of objects
  Destruct 008FF828: i_ = 2 p_ = 00C15440 *p = 3 // Dtor of a2. Note that a2.p_ = a1.p_
  Destruct 008FF838: i_ = 2 p_ = 00C15440 *p = -17891602 // Dtor of a1. a1.p_=a2.p_ points to garbage
  • The bit-wise copy of members is known as Shallow Copy
- **DEEP COPY:**
  • Now suppose we provide a user-defined copy constructor:

```
class A {
        int i_; // Non-pointer data member
        int* p_; // Pointer data member
        public:
                A(int i, int j) : i_(i), p_(new int(j)) { } // Init. with pointer to dynamically
        created object
                A(const A& a) : i_(a.i_), // Copy Constructor
                p_(new int(*a.p_)) { } // Allocation done and value copied - Deep Copy
                ~A() { cout << "Destruct " << this << ": "; // Object identity
```

```
        cout << "i_ = " << i_ << " p_ = " << p_ << " *p = " << *p_ << endl; // Object
    state
        delete p_; // Release resource
        }
};
```

• The output now is correct, as a1.p 6= a2.p points to the different int locations with the values *a1.p = *a2.p properly copied:

&a1 = 00B8F9E0 &a2 = 00B8F9D0 // Identities of objects
Destruct 00B8F9D0: i_ = 2 p_ = 00C95480 *p = 3 // Dtor of a2. a2.p_ is different from a1.p_
Destruct 00B8F9E0: i_ = 2 p_ = 00C95440 *p = 3 // Dtor of a1. Works correctly!

• This is known as Deep Copy where every member is copied properly. Note that:
◦ In every class, provide copy constructor to adopt to deep copy which is always safe
◦ Naturally, shallow copy is cheaper than deep copy. So some languages support variants as Lazy Copy or Copy-on-Write for efficiency

- **COPY ASSIGNMENT:**

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class String {
    public: char *str_; size_t len_;
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // Ctor
    String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // CCtor
    ~String() { free(str_); } // Dtor
    String& operator=(const String& s) { // Copy Assignment Operator
        free(str_); // Release existing memory
        str_ = strdup(s.str_); // Perform deep copy
        len_ = s.len_; // Copy data member of built-in type
        return *this; // Return object for chain assignment
    }
    void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
int main() {

    String s1 = "Football", s2 = "Cricket"; s1.print(); s2.print(); s2 = s1; s2.print();

}
```

Output:
(Football: 8)
(Cricket: 7)
(Football: 8)

- **SELF COPY:**

Unsafe----------

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class String {
```

```
        public: char *str_; size_t len_;
        String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // Ctor
        String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // CCtor
        ~String() { free(str_); } // Dtor
        String& operator=(const String& s) { // Copy Assignment Operator
                free(str_); // Release existing memory
                str_ = strdup(s.str_); // Perform deep copy
                len_ = s.len_; // Copy data member of built-in type
                return *this; // Return object for chain assignment
        }
        void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
int main() { String s1 = "Football", s2 = "Cricket"; s1.print(); s2.print(); s1 = s1; s1.print(); }
```

Output:

(Football: 8)
(Cricket: 7)
(????????: 8) // Garbage is printed. May crash too

Safe----------

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class String {
        public: char *str_; size_t len_;
                String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // Ctor
                String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // CCtor
                ~String() { free(str_); } // Dtor
                String& operator=(const String& s) { // Copy Assignment Operator
                        if (this != &s) { // Check if the source and destination are same
                                free(str_);
                                str_ = strdup(s.str_);
                                len_ = s.len_;
                        }
                        return *this;
                }
                void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
int main() { String s1 = "Football", s2 = "Cricket"; s1.print(); s2.print(); s1 = s1; s1.print(); }
```

Output:

(Football: 8)
(Cricket: 7)
(Football: 8)

- **SIGNATURE OF COPY ASSIGNMENT:**

  • For class MyClass, typical copy assignment operator will be:

```
MyClass& operator=(const MyClass& s) {
        if (this != &s) { // Check if the source and destination are same
                // Release resources held by *this
                // Copy members of s to members of *this
```

```
        }
        return *this; // Return object for chain assignment
}
```

• Signature of a Copy Assignment Operator can be one of:

```
MyClass& operator=(const MyClass& rhs); // Common. No change in Source
MyClass& operator=(MyClass& rhs); // Occasional. Change in Source
```

• The following Copy Assignment Operators are occasionally used:

```
MyClass& operator=(MyClass rhs);
const MyClass& operator=(const MyClass& rhs);
const MyClass& operator=(MyClass& rhs);
const MyClass& operator=(MyClass rhs);
MyClass operator=(const MyClass& rhs);
MyClass operator=(MyClass& rhs);
MyClass operator=(MyClass rhs);
```

CONST-NESS:

- **CONSTANT OBJECTS:**

• Like objects of built-in type, objects of user-defined types can also be made constant

• If an object is constant, none of its data members can be changed

• The type of the this pointer of a constant object of class, say, MyClass is:

```
// const Pointer to const Object
const MyClass * const this;
```

instead of

```
// const Pointer to non-const Object
MyClass * const this;
```

as for a non-constant object of the same class

• A constant objects cannot invoke normal methods of the class lest these methods change the object

Example:

```
#include <iostream>
using namespace std;
class MyClass {
        int myPriMember_;
        public: int myPubMember_;
                MyClass(int mPri, int mPub) : myPriMember_(mPri), myPubMember_(mPub)
        { }
                int getMember() { return myPriMember_; }
                void setMember(int i) { myPriMember_ = i; }
                void print() { cout << myPriMember_ << ", " << myPubMember_ << endl; }
};
int main() {
        const MyClass myConstObj(5, 6); // Constant object
        cout << myConstObj.getMember() << endl; // Error 1
        myConstObj.setMember(7); // Error 2
        myConstObj.myPubMember_ = 8; // Error 3
        myConstObj.print(); // Error 4
```

}

• To declare a constant member function, we use the keyword const between the function header and the body. Like:

void print() const { cout << myMember_ << endl; }

• Interesting, non-constant objects can invoke constant member functions

• Constant objects, however, can only invoke constant member functions

- **mutable MEMBERS:**

• While a constant data member is not changeable even in a non-constant object, a mutable data member is changeable in a constant object

◦ mutable is applicable only to data members and not to variables

◦ Reference data members cannot be declared mutable

◦ Static data members cannot be declared mutable

◦ const data members cannot be declared mutable

Example:

```
#include <iostream>
using namespace std;
class MyClass {
        int mem_;
        mutable int mutableMem_;
        public:
                MyClass(int m, int mm) : mem_(m), mutableMem_(mm) { }
                int getMem() const { return mem_; }
                void setMem(int i) { mem_ = i; }
                int getMutableMem() const { return mutableMem_; }
                void setMutableMem(int i) const { mutableMem_ = i; } // Okay to change
        mutable
};
int main() {
        const MyClass myConstObj(1, 2);
        cout << myConstObj.getMem() << endl;
        // myConstObj.setMem(3); // Error to invoke
        cout << myConstObj.getMutableMem() << endl;
        myConstObj.setMutableMem(4);
}
```


static MEMBERS:

- • A static data member

◦ is associated with class not with object

◦ is shared by all the objects of a class

◦ needs to be defined outside the class scope (in addition to the declaration within the class scope) to avoid linker error

◦ is constructed before main() starts and destructed after main() ends

◦ can be private / public

◦ can be accessed

. with the class-name followed by the scope resolution operator (::)

. as a member of any object of the class

- **Examples:**

```
using namespace std;
class MyClass {
        static int x; // Declare static
        public:
                void get() { x = 15; }
                void print() {
                        x = x + 10;
                        cout << "x =" << x << endl;
                }
};
int MyClass::x = 0; // Define static data member
int main() {
        MyClass obj1, obj2; // Have same x
        obj1.get(); obj2.get();
        obj1.print(); obj2.print();
}
```

Output:

x = 25 , x = 35

- **Order of Initialization:**

  • Order of initialization of static data members does not depend on their order in the definition of the class. It depends on the order their definition and initialization in the source

  Example:

```
#include <iostream>
#include <string>
using namespace std;
class Data {
        string id_;
        public:
                Data(const string& id) : id_(id)
                { cout << "Construct: " << id_ << endl; }
                ~Data()
                { cout << "Destruct: " << id_ << endl; }
};
class MyClass {
        static Data d2_; // Order of static members swapped
        static Data d1_;
};
Data MyClass::d1_("obj_1"); // Constructed 1st
Data MyClass::d2_("obj_2"); // Constructed 2nd
int main() { }
```

Output:

Construct: obj_1
Construct: obj_2
Destruct: obj_2
Destruct: obj_1

- **STATIC FUNCTION:**

  A static member function

◦ does not have this pointer – not associated with any object
◦ cannot access non-static data members
◦ cannot invoke non-static member functions
◦ can be accessed
. with the class-name followed by the scope resolution operator (::)
◦ **cannot be declared as const**

- **SINGLETON CLASS:**
  • Singleton is a creational design pattern
  ◦ ensures that only one object of its kind exists and
  ◦ provides a single point of access to it for any other code
  • **A class is called a Singleton if it satisfies the above conditions**
  Example:

```
#include <iostream>
using namespace std;
class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
        bool blackAndWhite_, bothSided_;
        Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
        { cout << "Printer constructed" << endl; }
        ~Printer() { cout << "Printer destructed" << endl; }
        public:
                static const Printer& printer(bool bw = false, bool bs = false) {
                        static Printer myPrinter(bw, bs); // The Singleton -- constructed the
                first time
                        return myPrinter;
                }
        void print(int nP) const { cout << "Printing " << nP << " pages" << endl; }
};
int main() {
        Printer::printer().print(10);
        Printer::printer().print(20);
}
```

Note: Concept used is every thing is private so you can't manipulate things by creating an object you need to use static and do things using class name hence only one instance.

friend FUNCTION AND friend CLASS:

- **friend FUNCTION:**

```
#include<iostream>
using namespace std;
class MyClass {
        int data_;
        public:
                MyClass(int i) : data_(i) { }
                friend void display(const MyClass& a);
};
void display(const MyClass& a) { // global function
        cout << "data = " << a.data_; // Okay
}
```

```
int main() {
        MyClass obj(10);
        display(obj);
}
```

- **Things on friend function:**

• A friend function of a class
◦ has access to the private and protected members of the class (breaks the encapsulation) in addition to public members
◦ must have its prototype included within the scope of the class prefixed with the keyword friend
◦ does not have its name qualified with the class scope
◦ is not called with an invoking object of the class
◦ can be declared friend in more than one classes
• A friend function can be a
◦ global function
◦ a member function of a class
◦ a function template

Example:

```
#include <iostream>
using namespace std;
class Node; // Forward declaration
class List {
        Node *head; // Head of the list
        Node *tail; // Tail of the list
        public:
                List(Node *h = 0): head(h), tail(h) { }
                void display();
                void append(Node *p);
};
class Node {
        int info; // Data of the node
        Node *next; // Ptr. to next node
        public:
                Node(int i): info(i), next(0) { }
                friend void List::display();
                friend void List::append(Node *);
};
void List::display() { // friend of Node
        Node *ptr = head;
        while (ptr) { cout << ptr->info << " ";
                ptr = ptr->next;
        }
}
void List::append(Node *p) { // friend of Node
        if (!head) head = tail = p;
        else {
                tail->next = p;
                tail = tail->next;
```

```
        }
    }
    int main() {
        List l; // Init. null list
        Node n1(1), n2(2), n3(3); // Few nodes
        l.append(&n1); // Add nodes to list
        l.append(&n2);
        l.append(&n3);
        l.display(); // Show list
    }
```

- **friend CLASS:**

  • A friend class of a class

  ◦ has access to the private and protected members of the class (breaks the encapsulation) in addition to public members

  ◦ does not have its name qualified with the class scope (not a nested class)

  ◦ can be declared friend in more than one classes

  • A friend class can be a

  ◦ class

  ◦ class template

  Example:

```
#include <iostream>
using namespace std;
class Node; // Forward declaration
class List {
        Node *head; // Head of the list
        Node *tail; // Tail of the list
        public:
                List(Node *h = 0): head(h), tail(h) { }
                void display();
                void append(Node *p);
};
class Node {
        int info; // Data of the node
        Node *next; // Ptr to next node
        public:
                Node(int i): info(i), next(0) { }
                // friend void List::display();
                // friend void List::append(Node *);
                friend class List;
};
void List::display() {
        Node *ptr = head;
        while (ptr) { cout << ptr->info << " ";
                ptr = ptr->next;
        }
}
void List::append(Node *p) {
        if (!head) head = tail = p;
```

```
        else {
            tail->next = p;
            tail = tail->next;
        }
}
int main() {
        List l; // Init null list
        Node n1(1), n2(2), n3(3); // Few nodes
        l.append(&n1); // Add nodes to list
        l.append(&n2);
        l.append(&n3);
        l.display(); // Show list
}
```

OVERLOADING OPERATOR FOR USER-DEFINED TYPES:

- **Non-Member Operator Function:**
  • A non-member operator function may be a
  ◦ Global Function
  ◦ friend Function
  • Binary Operator:

```
MyType a, b; // An enum, struct or class
MyType operator+(const MyType&, const MyType&); // Global
friend MyType operator+(const MyType&, const MyType&); // Friend
```

  • Unary Operator:

```
MyType operator++(const MyType&); // Global
friend MyType operator++(const MyType&); // Friend
```

  • Note: The parameters may not be constant and may be passed by value. The return may also
  be by reference and may be constant
  • Examples:

Operator Expression	Operator Function
a + b	operator+(a, b)
a = b	operator=(a, b)
++a	operator++(a)
a++	operator++(a, int) Special Case
c = a + b	operator=(c, operator+(a, b))

Example:

```
#include <iostream>
using namespace std;
class Complex { // Private data members
        double re, im;
        public:
            Complex(double a=0.0, double b=0.0): re(a), im(b) { } ~Complex() { }
            void display();
            double real() { return re; }
            double img() { return im; }
            double set_real(double r) { re = r; }
            double set_img(double i) { im = i; }
```

```
};
void Complex::display() {
        cout << re << " +j " << im << endl;
}
Complex operator+(Complex &t1, Complex &t2) {
        Complex sum;
        sum.set_real(t1.real() + t2.real());
        sum.set_img(t1.img() + t2.img());
        return sum;
}
int main() {
        Complex c1(4.5, 25.25), c2(8.3, 10.25), c3;
        cout << "1st complex No:"; c1.display();
        cout << "2nd complex No:"; c2.display();
        c3 = c1 + c2; // Overload operator +
        cout << "Sum = "; c3.display();
        }
```

- **Member Operator Function:**

• Binary Operator:
MyType a, b; // MyType is a class
MyType operator+(const MyType&); // Operator function
• The left operand is the invoking object – right is taken as a parameter
• Unary Operator:
MyType operator-(); // Operator function for Unary minus
MyType operator++(); // For Pre-Incrementer
MyType operator++(int); // For post-Incrementer
• The only operand is the invoking object
• Note: The parameters may not be constant and may be passed by value. The return may also
be by reference and may be constant
• Examples:

Operator Expression	Operator Function
a + b	a.operator+(b)
a = b	a.operator=(b)
++a	a.operator++()
a++	a.operator++(int) // Special Case
c = a + b	c.operator =(a.operator+(b))

Example:

```
#include <iostream>
using namespace std;
class Complex { // Private data members
        double re, im;
        public:
                Complex(double a=0.0, double b=0.0): re(a), im(b) { } ~Complex() { }
                void display();
                Complex operator+(const Complex &c) {
                        Complex r;
```

```
                    r.re = re + c.re;
                    r.im = im + c.im;
                    return r;
                }
};

void Complex::display() {
        cout << re;
        cout << " +j " << im << endl;
}
int main() {
        Complex c1(4.5, 25.25), c2(8.3, 10.25), c3;
        cout << "1st complex No:";
        c1.display();
        cout << "2nd complex No:";
        c2.display();
        c3 = c1 + c2; // Overloaded operator +
        cout << "Sum = ";
        c3.display();
        return 0;
}
```

- **ISSUES IN OPERATOR OVERLOADING:**

• Consider a Complex class. We have learnt how to overload operator+ to add two Complex numbers:

```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
d3 = d1 + d2; // d3 = 4.1 +j 6.5
```

• Now we want to extend the operator so that a Complex number and a real number (no imaginary part) can be added together:

```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
d3 = d1 + 6.2; // d3 = 8.7 +j 3.2
d3 = 4.2 + d2; // d3 = 5.8 +j 3.3
```

• We show why global operator function is not good for this

• We show why member operator function cannot do this

• We show how friend function achieves this

With Global Function:

```
#include <iostream>
using namespace std;
class Complex {
        public: double re, im;
                explicit Complex(double r = 0, double i = 0): re(r), im(i) { } // No implicit
        conversion is allowed
                void disp() { cout << re << " +j " << im << endl; }
};
Complex operator+(const Complex &a, const Complex &b) { // Overload 1
        return Complex(a.re + b.re, a.im + b.im);
}
Complex operator+(const Complex &a, double d) { // Overload 2
        Complex b(d); return a + b; // Create temporary object and use Overload 1
```

```
}
Complex operator+(double d, const Complex &b) { // Overload 3
        Complex a(d); return a + b; // Create temporary object and use Overload 1
}
int main() {
        Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
        d3 = d1 + d2; d3.disp(); // d3 = 4.1 +j 6.5. Overload 1
        d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2
        d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3. Overload 3
}
```

- **Works fine with global functions - 3 separate overloading are provided**
- **A bad solution as it breaks the encapsulation – as discussed in Module 18**
- **Let us try to use member function**
- **Note: A simpler solution uses Overload 1 and implicit casting (for this we need to remove explicit before constructor).**

But that too breaks encapsulation. We discuss this when we take up cast operators

With Member Function:

```
#include <iostream>
using namespace std;
class Complex {
        double re, im;
        public:
                explicit Complex(double r = 0, double i = 0) : re(r), im(i) { } // No implicit
        conversion is allowed
                void disp() { cout << re << " +j " << im << endl; }
                Complex operator+(const Complex &a) { // Overload 1
                return Complex(re + a.re, im + a.im);
}
Complex operator+(double d) { // Overload 2
        Complex b(d); // Create temporary object
        return *this + b; // Use Overload 1
        }
};
int main() {
        Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
        d3 = d1 + d2; d3.disp(); // d3 = 4.1 +j 6.5. Overload 1
        d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2
        //d3 = 4.2 + d2; // Overload 3 is not possible - needs an object on left
        //d3.disp();
}
```

- **Overload 1 and 2 works**
- **Overload 3 cannot be done because the left operand is double – not an object**
- **Let us try to use friend function**
- **Note: This solution too avoids the feature of cast operators**

With Friend Function:

```
#include <iostream>
using namespace std;
class Complex {
```

```
        double re, im;
        public:
            explicit Complex(double r = 0, double i = 0) : re(r), im(i) { } // No implicit
        conversion is allowed
            void disp() { cout << re << " +j " << im << endl; }
            friend Complex operator+(const Complex &a, const Complex &b) {
                return Complex(a.re + b.re, a.im + b.im);
            }
            friend Complex operator+(const Complex &a, double d) { // Overload 2
                Complex b(d); // Create temporary object
                return a + b; // Use Overload 1
            }
            friend Complex operator+(double d, const Complex &b) { // Overload 3
                Complex a(d); // Create temporary object
                return a + b; // Use Overload 1
            }
};
int main() {
        Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
        d3 = d1 + d2; d3.disp(); // d3 = 4.1 +j 6.5. Overload 1
        d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2. Overload 2
        d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3. Overload 3
}
```

• Works fine with friend functions - 3 separate overloading are provided and Preserves the encapsulation too

• Note: A simpler solution uses only Overload 1 and implicit casting (for this we need to remove explicit before
constructor) will be discussed when we take up cast operators

Examples:

a)

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <cstring>
using namespace std;
class MyStr {
        const char *name_;
        public:
            explicit MyStr(const char *s) : name_(strdup(s)) { } ~MyStr() { free((void
        *)name_); }
            friend bool operator==(const MyStr& s1, const MyStr& s2) { return
        !strcmp(s1.name_, s2.name_); } // 1
            friend bool operator==(const MyStr& s1, const string& s2) { return
        !strcmp(s1.name_, s2.c_str()); } // 2
            friend bool operator==(const string& s1, const MyStr& s2) { return
        !strcmp(s1.c_str(), s2.name_); } // 3
};
int main() {
```

```
        MyStr mS1("red"), mS2("red"), mS3("blue"); string sS1("red"), sS2("red"),
sS3("blue");
        if (mS1 == mS2) cout << "Match "; else cout << "Mismatch ";
        if (mS1 == mS3) cout << "Match "; else cout << "Mismatch ";
        if (mS1 == sS2) cout << "Match "; else cout << "Mismatch ";
        if (mS1 == sS3) cout << "Match "; else cout << "Mismatch ";
        if (sS1 == mS2) cout << "Match "; else cout << "Mismatch ";
        if (sS1 == mS3) cout << "Match "; else cout << "Mismatch ";
        if (sS1 == sS2) cout << "Match "; else cout << "Mismatch ";
        if (sS1 == sS3) cout << "Match "; else cout << "Mismatch ";
}
```

b)

```
#include <iostream>
using namespace std;
class Complex {
        public:
                double re, im;
                Complex(double r = 0, double i = 0): re(r), im(i) { }
};
ostream& operator<<(ostream& os, const Complex &a) {
        os << a.re << " +j " << a.im << endl;
        return os;
}
istream& operator>>(istream& is, Complex &a) {
        is >> a.re >> a.im;
        return is;
}
int main() {
        Complex d;
        cin >> d;
        cout << d;
}
```

c)

```
#include <iostream>
using namespace std;
class Complex {
        double re, im;
        public:
                Complex(double r = 0, double i = 0): re(r), im(i) { }
                friend ostream& operator<<(ostream& os, const Complex &a);
                friend istream& operator>>(istream& is, Complex &a);
};
friend ostream& operator<<(ostream& os, const Complex &a) {
        os << a.re << " +j " << a.im << endl;
        return os;
}
```

```
friend istream& operator>>(istream& is, Complex &a) {
        is >> a.re >> a.im;
        return is;
}
int main() {
        Complex d;
        cin >> d;
        cout << d;
}
```

NAMESPACE:

- **namespace FUNDEMENTAL:**

Example:

```
#include <iostream>
using namespace std;
namespace MyNameSpace {
        int myData;
        void myFunction() { cout << "MyNameSpace myFunction" << endl; }
        class MyClass {
                int data;
                public:
                        MyClass(int d) : data(d) { }
                                void display() { cout << "MyClass data = " << data << endl; }
        };
}
int main() {
        MyNameSpace::myData = 10;
        cout << "MyNameSpace::myData = " << MyNameSpace::myData << endl;
        MyNameSpace::myFunction();
        MyNameSpace::MyClass obj(25);
        obj.display();
}
```

• A name in a namespace is prefixed by the name of it

• Beyond scope resolution, all namespace items are treated as global

- **namespace FEATURES:**

(Nested)----------

```
#include <iostream>
using namespace std;
int data = 0; // Global name ::
namespace name1 {
        int data = 1; // In namespace name1
        namespace name2 {
                int data = 2; // In nested namespace name1::name2
        }
}
int main() {
        cout << data << endl; // 0
```

```
        cout << name1::data << endl; // 1
        cout << name1::name2::data << endl; // 2
        return 0;
}
```

(using namespace)----------

```
#include <iostream>
using namespace std;
namespace name1 {
        int v11 = 1;
        int v12 = 2;
}
namespace name2 {
        int v21 = 3;
        int v22 = 4;
}
using namespace name1; // All symbols of namespace name1 will be available
using name2::v21; // Only v21 symbol of namespace name2 will be available
int main() {
        cout << v11 << endl; // name1::v11
        cout << name1::v12 << endl; // name1::v12
        cout << v21 << endl; // name2::v21
        cout << name2::v21 << endl; // name2::v21
        cout << v22 << endl; // Treated as undefined
}
```

(global namespace) ----------

```
#include <iostream>
using namespace std;
int data = 0; // Global Data
namespace name1 {
        int data = 1; // namespace Data
}
int main() {
using name1::data;
        cout << data << endl; // 1 // name1::data -- Hides global data
        cout << name1::data << endl; // 1
        cout << ::data << endl; // 0 // ::data -- global data
}
```

(new declerations)----------

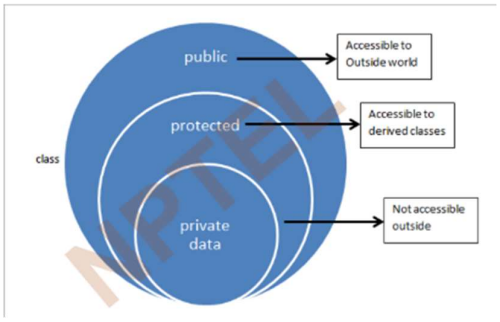
```
#include <iostream>
using namespace std;
namespace open // First definition
{ int x = 30; }
namespace open // Additions to the last definition
{ int y = 40; }
int main() {
        using namespace open; // Both x and y would be available
        x = y = 20;
        cout << x << " " << y ;
```

}

INHERITANCE SEMANTICS:

I think you guys know about inheritance if you are already familiar with other object-oriented programming languages so no recap here.

DATA MEMBER & MEMBER FUNCTION: OVERRIDE & OVERLOAD:



• • Derived ISA Base
• **Data Members**
◦ Derived class inherits all data members of Base class
◦ Derived class may add data members of its own
• Member Functions
◦ Derived class inherits all member functions of Base class
◦ Derived class may override a member function of Base class by redefining it with the same signature
◦ Derived class may overload a member function of Base class by redefining it with the same name;
but different signature
◦ Derived class may add new member functions
• Access Specification
◦ Derived class cannot access private members of Base class
◦ Derived class can access protected members of Base class
• Construction-Destruction
◦ A constructor of the Derived class must first call a constructor of the Base class to construct the
Base class instance of the Derived class
◦ The destructor of the Derived class must call the destructor of the Base class to destruct the Base
class instance of the Derived class

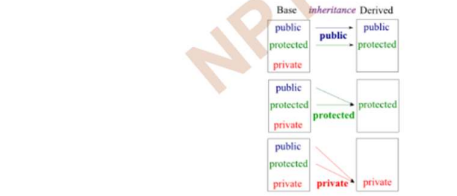
- • Derived ISA Base
  • **Member Functions**
  ◦ Derived class inherits all member functions of Base class
  . Note: Derived class does not inherit the Constructors and Destructor of Base class but must have access to them
  ◦ Derived class may override a member function of Base class by redefining it with the same signature
  ◦ Derived class may overload a member function of Base class by redefining it with the same name; but different signature
  ◦ Derived class may add new member functions
  • **Static Member Functions**
  ◦ **Derived class does not inherit the static member functions of Base class**
  • **Friend Functions**
  ◦ **Derived class does not inherit the friend functions of Base class**
- **Example:**

```
class B {
        public: // Base Class
                void f(int);
                void g(int i);
};
class D: public B {
        public: // Derived Class
                // Inherits B::f(int)
                void f(int); // Overrides B::f(int)
                void f(string&); // Overloads B::f(int)
                // Inherits B::g(int)
                void h(int i); // Adds D::h(int)
};
B b;
D d;
b.f(1);
b.g(2);
d.f(3);
d.g(4);
d.f("red");
d.h(5);
```

CONSTRUCTOR AND DISTRUCTOR: OBJECT LIFETIME:

- Visibility Matrix

Visibility \ Inheritance	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private



- **RULES:**
- Derived ISA Base
- **Constructor-Destructor**
  - Derived class does not inherit the Constructors and Destructor of Base class but must have access to them
  - Derived class must provide its own Constructors and Destructor
  - Derived class cannot override or overload a Constructor or the Destructor of Base class
- **Construction-Destruction**
  - A constructor of the Derived class must first call a constructor of the Base class to construct the Base class instance of the Derived class
  - **The destructor of the Derived class must call the destructor of the Base class to destruct the Base class instance of the Derived class**

Example:

```
class B {
        protected: int data_;
        public:
                B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }
                ~B() { cout << "B::~B(): " << data_ << endl; }
                // ...
};
```

```
class D: public B {
        int info_;
        public:
                D(int d, int i) : B(d), info_(i) // ctor-1: Explicit construction of Base
                { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }
                D(int i) : info_(i) // ctor-2: Default construction of Base
                { cout << "D::D(int): " << data_ << ", " << info_ << endl; }
                ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
                // ...
};
B b(5);
D d1(1, 2); // ctor-1: Explicit construction of Base
D d2(3);
```

- **OBJECT LIFE TIME:**

```
class B {
        protected: int data_;
        public:
                B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }
                ~B() { cout << "B::~B(): " << data_ << endl; }
                // ...
};
class D: public B {
        int info_;
        public:
                D(int d, int i) : B(d), info_(i) // ctor-1: Explicit construction of Base
                { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }
                D(int i) : info_(i) // ctor-2: Default construction of Base
                { cout << "D::D(int): " << data_ << ", " << info_ << endl; }
                ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
                // ...
};
B b;
D d1(1, 2); // ctor-1: Explicit construction of Base
D d2(3); // ctor-2: Default construction of Base
```

Output:

Construction O/P

B::B(int): 0 // Object b
B::B(int): 1 // Object d1
D::D(int, int): 1, 2 // Object d1
B::B(int): 0 // Object d2
D::D(int): 0, 3 // Object d2

Destruction O/P

D::~D(): 0, 3 // Object d2
B::~B(): 0 // Object d2
D::~D(): 1, 2 // Object d1
B::~B(): 1 // Object d1
B::~B(): 0 // Object b

- First construct base class object, then derived class object
- First destruct derived class object, then base class object

TYPE CASTING – 1:

Casting can be implicit or explicit. Casting in built-in types does not invoke any conversion function. It only re-interprets the binary representation

- ***Example 1:***

```
int i = 3;
double d = 2.5, *p = &d;
d = i; // implicit: int to double
i = d; // implicit: warning: '=' : conversion from 'double' to 'int': possible loss of data
d = (double)i; // explicit: int to double
i = (int)d; // explicit: double to int
i = p; // error: '=' : cannot convert from 'double *' to 'int'
i = (int)p; // explicit: double * to int
```

• *Implicit casting between different pointer types is not allowed*

• *Any pointer can be implicitly cast to void* (with loss of type); but void* cannot be implicitly cast to any pointer type*

• *Conversion between array and corresponding pointer is not type casting – these are two different syntactic forms for accessing the same data*

- ***Example 2:***

```
int i = 1, *p = &i, a[10]; double d = 1.1, *q = &d; void *r;
q = p; // error: cannot convert 'int*' to 'double*'
p = q; // error: cannot convert 'double*' to 'int*'
q = (double*)p; // Okay
p = (int*)q; // Okay
r = p; // Okay to convert from 'int*' to 'void*'
p = r; // error: invalid conversion from 'void*' to 'int*'
p = (int*)r; // Okay
p = a; // Okay by array pointer duality. p[i], a[i], *(p+i), *(a+i) are equivalent
a = p; // error: incompatible types in assignment of 'int*' to 'int[10]'
```

• *Implicit casting between pointer type and numerical type is not allowed*

• *However, explicit casting between pointer and integral type (int or long etc.) is a common practice to support various tasks like serialization (save a file) and de-serialization (open a file)*

• *Care should be taken with these explicit cast to ensure that the integral type is of the same size as of the pointer*.

- ***Example 3:***

```
int i, *p = 0; long j;
// sizeof(i) = sizeof(int) = 4
```

```
// sizeof(j) = sizeof(long) = 8
// sizeof(p) = sizeof(int*) = sizeof(void*) = 8
i = p; // error: invalid conversion from 'int*' to 'int'
p = i; // error: invalid conversion from 'int' to 'int*'
i = (int)p; // error: cast from 'int*' to 'int' loses precision
p = (int*)i; // warning: cast to pointer from integer of different size
j = (long)p; // Okay
p = (int*)j; // Okay
```

• *(Implicit) Casting between unrelated classes is not permitted*

• *Forced Casting between unrelated classes is dangerous*

- ***Example 4:***

```
class A { int i; };
class B { double d; };
A a;
B b;
A *p = &a;
B *q = &b;
a = b; // error: binary '=' : no operator which takes a right-hand operand of type 'B'
a = (A)b; // error: 'type cast' : cannot convert from 'B' to 'A'
b = a; // error: binary '=' : no operator which takes a right-hand operand of type 'A'
b = (B)a; // error: 'type cast' : cannot convert from 'A' to 'B'
p = q; // error: '=' : cannot convert from 'B *' to 'A *'
q = p; // error: '=' : cannot convert from 'A *' to 'B *'
p = (A*)&b; // explicit on pointer: type cast is okay for the compiler
q = (B*)&a; // explicit on pointer: type cast is okay for the compiler
cout << p->i << endl; // prints -858993459: GARBAGE
cout << q->d << endl; // prints -9.25596e+061: GARBAGE
```


Casting on a hierarchy is permitted in a limited sense

• *Up-Casting is safe*

• *Down-Casting is risky*

- ***Example 5:***

```
class A { };
class B : public A { };
A *pa = 0;
B *pb = 0;
void *pv = 0;
pa = pb; // UPCAST: Okay
pb = pa; // DOWNCAST: error: '=' : cannot convert from 'A *' to 'B *'
pv = pa; // Okay, but lose the type for A * to void *
pv = pb; // Okay, but lose the type for B * to void *
pa = pv; // error: '=' : cannot convert from 'void *' to 'A *'
pb = pv; // error: '=' : cannot convert from 'void *' to 'B *'
```

note: you can down cast by explicit casting but it will print garbage values.

TYPE BINDING:

• The static type of the object is the type declared for the object while writing the code

• Compiler sees static type

• The dynamic type of the object is determined by the type of the object to which it refers at run-time

• Compiler does not see dynamic type

◦ Function pointers, Virtual functions are examples of late binding

- ***Example:***

```
class A { };
class B : public A { };
int main() {
        A *p;
        p = new B;      // Static type of p is A*
                        // Dynamic type of p is B*

}
```

- **STATIC BINDING:**

```
#include<iostream>
using namespace std;
class B { public:
        void f() { }
};
class D : public B { public:
        void f() { }
};
int main() {
        B b; D d;
        b.f();    // B::f()
        d.f();    // D::f() ----- Overridden
                  // masks the base class function
}
```

- **using Construct:**

```
#include<iostream>
using namespace std;
class A { public:
        void f() { }
};
class B : public A { public:
        // To overload, rather than hide the base class function f(),
        // it is introduced into the scope of B with a using declaration
        using A::f;
        void f(int) { } // Overloads f()
};
```

```
int main() {
        B b; // function calls resolved at compile time
        b.f(3); // B::f(int)
        b.f(); // A::f()
}
```

- **DYNAMIC BINDING:**
  (VIRTUAL METHOD)

```
#include<iostream>
using namespace std;
class B { public:
        virtual void f() { }
};
class D : public B { public:
        virtual void f() { }
};
int main() {
        B b;
        D d;
        B *p;
        p = &b; p->f(); // B::f()
        p = &d; p->f(); // D::f()
}
```

• *Dynamic binding is possible only for pointer and reference data types and for member functions that are declared as virtual in the base class*

- ***Example:***

```
#include <iostream>
using namespace std;
class A { public:
    void f()         { cout << "A::f()" << endl; } // Non-Virtual
    virtual void g() { cout << "A::g()" << endl; } // Virtual
    void h()         { cout << "A::h()" << endl; } // Non-Virtual
};
class B : public A { public:
    void f()         { cout << "B::f()" << endl; } // Non-Virtual
    void g()         { cout << "B::g()" << endl; } // Virtual
    virtual void h() { cout << "B::h()" << endl; } // Virtual
};
class C : public B { public:
    void f()         { cout << "C::f()" << endl; } // Non-Virtual
    void g()         { cout << "C::g()" << endl; } // Virtual
    void h()         { cout << "C::h()" << endl; } // Virtual
};
```

```
int main() {
    B *q = new C; A *p = q;

    p->f();
    p->g();
    p->h();

    q->f();
    q->g();
    q->h();
}
```

```
A::f()
C::g()
A::h()
B::f()
C::g()
C::h()
```

- **Virtual Destructor:**

```
#include <iostream>
using namespace std;
class B { int data_; public:
        B(int d) :data_(d) { cout << "B()" << endl; }
        virtual ~B() { cout << "~B()" << endl; } // Destructor made virtual
        virtual void Print() { cout << data_; }
};
class D: public B { int *ptr_; public:
        D(int d1, int d2) :B(d1), ptr_(new int(d2)) { cout << "D()" << endl; }
        ~D() { cout << "~D()" << endl; delete ptr_; }
        void Print() { B::Print(); cout << " " << *ptr_; }
};
int main() {
        B *p = new B(2);
        B *q = new D(3, 5);
        p->Print(); cout << endl;
        q->Print(); cout << endl;
        delete p;
        delete q;
}
```

Output:

B()
B()
D()
2
3 5
~B()
~D()
~B()
Destructor of d (type D) is called!

- **Abstract Base Class:**
  A class containing at least one Pure Virtual Function is called an Abstract Base Class
  A Pure Virtual Function may have a body!
  Instances of Abstract class cant be created.
  Example:

```
#include <iostream>
using namespace std;
class Shapes { public: // Abstract Base Class
        virtual void draw() = 0 // Pure Virtual Function
        { cout << "Shapes: Init Brush" << endl; }
};
class Polygon: public Shapes { public: // Concrete Class
        void draw() { Shapes::draw(); cout << "Polygon: Draw by Triangulation" << endl; }
};
class ClosedConics: public Shapes { public: // Abstract Base Class
        // draw() inherited - Pure Virtual
};
class Triangle: public Polygon { public: // Concrete Class
```

```
        void draw() { Shapes::draw(); cout << "Triangle: Draw by Lines" << endl; }
};
class Quadrilateral: public Polygon { public: // Concrete Class
        void draw() { Shapes::draw(); cout << "Quadrilateral: Draw by Lines" << endl; }
};
class Circle: public ClosedConics { public: // Concrete Class
        void draw() { Shapes::draw(); cout << "Circle: Draw by Breshenham Algorithm" <<
endl; }
};
class Ellipse: public ClosedConics { public: // Concrete Class
        void draw() { Shapes::draw(); cout << "Ellipse: Draw by ..." << endl; }
};
int main() {
        Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
        for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i)
                arr[i]->draw();
}
```

Output:

Shapes: Init Brush
Triangle: Draw by Lines
Shapes: Init Brush
Quadrilateral: Draw by Lines
Shapes: Init Brush
Circle: Draw by Breshenham Algorithm
Shapes: Init Brush
Ellipse: Draw by ...

Binding: Exercise 1: Solution

```
// Class Definitions
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};
```

```
// Application Codes
A a;
B b;
C c;

A *pA;
B *pB;
```

Invocation	Initialization		
	pA = &a;	pA = &b;	pA = &c;
pA->f(2);	A::f	B::f	B::f
pA->g(3.2);	A::g	A::g	C::g
pA->h(&a);	A::h	A::h	A::h
pA->h(&b);	A::h	A::h	A::h

Binding: Exercise 2: Solution

```
// Class Definitions
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};
```

```
// Application Codes
A a;
B b;
C c;

A *pA;
B *pB;
```

Invocation	Initialization		
	pB = &a;	pB = &b;	pB = &c;
pB->f(2);	Error	B::f	B::f
pB->g(3.2);	Downcast	A::g	C::g
pB->h(&a);	(A *) to	No conversion (A *) to (B *)	
pB->h(&b);	(B *)	B::h	C::h

----------Virtual Function Pointer Table Not completed Yet

TYPE CASTING 2:

- **const_cast Operator:**
  const cast converts between types with different cv-qualification
  Only const cast may be used to cast away (remove) const-ness or volatility
  Usually does not perform computation or change value
  Example 1:
  #include <iostream>

```
using namespace std;
class A {
        int i_;
        public: A(int i) : i_(i) { }
        int get() const { return i_; }
        void set(int j) { i_ = j; }
};
void print(char * str) { cout << str; }
int main() {
        const char * c = "sample text";
        // print(c); // error: 'void print(char *)': cannot convert argument 1 from 'const char *' to 'char *'
        print(const_cast<char *>(c)); // Okay
        print((char *)(c)); // C-Style Cast
        const A a(1);
        a.get();
        // a.set(5); // error: 'void A::set(int)': cannot convert 'this' pointer from 'const A' to 'A &'
        const_cast<A&>(a).set(5); // Okay
        // const_cast<A>(a).set(5); // error: 'const_cast': cannot convert from 'const A' to 'A'
}
```

Example 2:

```
#include <iostream>
struct type { type(): i(3) { }
        void m1(int v) const {
                //this->i = v; // error C3490: 'i' cannot be modified -- accessed through a const object
                const_cast<type*>(this)->i = v; // Okay as long as the type object isn't const
        }
        int i;
};
int main() {
        int i = 3; // i is not declared const
        const int& cref_i = i; const_cast<int&>(cref_i) = 4; // Okay: modifies i
        std::cout << "i = " << i << '\n';
        type t; // note, if this is const type t;, then t.m1(4); may be undefined behavior
        t.m1(4);
        std::cout << "type::i = " << t.i << '\n';
        const int j = 3; // j is declared const
        int* pj = const_cast<int*>(&j); *pj = 4; // undefined behavior! Value of j and *pj may differ
        std::cout << j << " " << *pj << std::endl;
        void (type::*mfp)(int) const = &type::m1; // pointer to member function
        //const_cast<void(type::*)(int)>(mfp); // error C2440: 'const_cast': cannot convert from
        // 'void (__thiscall type::* )(int) const' to
        // 'void (__thiscall type::* )(int)' const_cast does not work
}
```

Output:

i = 4
type::i = 4
3 4

TYPE CASTING 3:

- **static_cast Operator:**
  • static cast performs all conversions allowed implicitly (not only those with pointers to classes), and also the opposite of these. It can:
  ◦ Convert from void* to any pointer type
  ◦ Convert integers, floating-point values to enum types
  ◦ Convert one enum type to another enum type
  • static cast can perform conversions between pointers to related classes:
  ◦ Not only up-casts, but also down-casts
  ◦ No checks are performed during run-time to guarantee that the object being converted is in fact a full object of the destination type
  • Additionally, static cast can also perform the following:
  ◦ Explicitly call a single-argument constructor or a conversion operator – The User-Defined Cast
  ◦ Convert to rvalue references
  ◦ Convert enum values into integers or floating-point values
  ◦ Convert any type to void, evaluating and discarding the value

  Example 1:

```
#include <iostream>
using namespace std;
int main() {
        // Built-in Types
        int i = 2; long j; double d = 3.7; int *pi = &i; double *pd = &d; void *pv = 0;
        i = d; // implicit -- warning
        i = static_cast<int>(d); // static_cast -- okay
        i = (int)d; // C-style -- okay
        d = i; // implicit -- okay
        d = static_cast<double>(i); // static_cast -- okay
        d = (double)i; // C-style -- okay
        pv = pi; // implicit -- okay
        pi = pv; // implicit -- error
        pi = static_cast<int*>(pv); // static_cast -- okay
        pi = (int*)pv; // C-style -- okay
        j = pd; // implicit -- error
        j = static_cast<long>(pd); // static_cast -- error
        j = (long)pd; // C-style -- okay: sizeof(long) = 8 = sizeof(double*)

        // RISKY: Should use reinterpret_cast

        i = (int)pd; // C-style -- error: sizeof(int) = 4 != 8 = sizeof(double*)

        // Refer to Module 26 for details

}
```

  Example 2:

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

```
// Class Hierarchy
class A { };
class B: public A { };
int main() {
        A a;
        B b;
        // UPCAST
        A *p = 0;
        p = &b; // implicit -- okay
        p = static_cast<A*>(&b); // static_cast -- okay
        p = (A*)&b; // C-style -- okay
        // DOWNCAST
        B *q = 0;
        q = &a; // implicit -- error
        q = static_cast<B*>(&a); // static_cast -- okay: RISKY: Should use dynamic_cast
        q = (B*)&a; // C-style -- okay
}
```

Example 3:

```
#include <iostream>
using namespace std;
// Un-related Types
class B;
class A { public:
        A(int i = 0) { cout << "A::A(i)\n"; }
        A(const B&) { cout << "A::A(B&)\n"; } // both needed for casting
};
class B { };
int main() {
        A a; B b;
        int i = 5;
        // B ==> A
        a = b; // Uses A::A(B&)
        a = static_cast<A>(b); // Uses A::A(B&)
        a = (A)b; // Uses A::A(B&)
        // int ==> A
        a = i; // Uses A::A(int)
        a = static_cast<A>(i); // Uses A::A(int)
        a = (A)i; // Uses A::A(int)
}
```

Example 4:

```
#include <iostream>
using namespace std;
// Un-related Types
class B;
class A { int i_; public:
        A(int i = 0) : i_(i) { cout << "A::A(i)\n"; }
        operator int() { cout << "A::operator int()\n"; return i_; }
};
```

```
class B { public:
        operator A() { cout << "B::operator A()\n"; return A(); }
};
int main() { A a; B b; int i = 5;
        // B ==> A
        a = b; // B::operator A()
        a = static_cast<A>(b); // B::operator A()
        a = (A)b; // B::operator A()
        // A ==> int
        i = a; // A::operator int()
        i = static_cast<int>(a); // A::operator int()
        i = (int)a; // A::operator int()
}
```

- **Reinterpret_cast Operator:**
  • reinterpret cast converts any pointer type to any other pointer type, even of unrelated classes
  • The operation result is a simple binary copy of the value from one pointer to the other
  • All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked
  • It can also cast pointers to or from integer types
  • The format in which this integer value represents a pointer is platform-specific
  • The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as intptr t), is guaranteed to be able to be cast back to a valid pointer (Refer to Module 26)
  • The conversions that can be performed by reinterpret cast but not by static cast are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable

  Example:

```
#include <iostream>
using namespace std;
class A { };
class B { };
int main() {
        long i = 2;
        double d = 3.7;
        double *pd = &d;
        i = pd; // implicit -- error
        i = reinterpret_cast<long>(pd); // reinterpret_cast -- okay
        i = (long)pd; // C-style -- okay
        cout << pd << " " << i << endl;
        A *pA;
        B *pB;
        pA = pB; // implicit -- error
        pA = reinterpret_cast<A*>(pB); // reinterpret_cast -- okay
        pA = (A*)pB; // C-style -- okay
}
```

- **dynamic_cast Operator:**
  • dynamic cast can only be used with pointers and references to classes (or with void*)

• Its purpose is to ensure that the result of the type conversion points to a valid complete object
of the destination pointer type
• This naturally includes pointer upcast (converting from pointer-to-derived to pointer-to-base),
in the same way as allowed as an implicit conversion
• But dynamic cast can also downcast (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if-and-only-if the pointed object is a valid complete object of the target type
• If the pointed object is not a valid complete object of the target type, dynamic cast returns a
null pointer
• If dynamic cast is used to convert to a reference type and the conversion is not possible, an exception of type bad cast is thrown instead
• dynamic cast can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a void* pointer

Example 1:

```
#include <iostream>
using namespace std;
class A { public: virtual ~A() { } };
class B: public A { };
class C { public: virtual ~C() { } };
int main() { A a; B b; C c;
        B* pB = &b; A *pA = dynamic_cast<A*>(pB);
        cout << pB << " casts to " << pA << ": Up-cast: Valid" << endl;
        pA = &b; pB = dynamic_cast<B*>(pA);
        cout << pA << " casts to " << pB << ": Down-cast: Valid" << endl;
        pA = &a; pB = dynamic_cast<B*>(pA);
        cout << pA << " casts to " << pB << ": Down-cast: Invalid" << endl;
        pA = (A*)&c; C *pC = dynamic_cast<C*>(pA);
        cout << pA << " casts to " << pC << ": Unrelated-cast: Invalid" << endl;
        pA = 0; pC = dynamic_cast<C*>(pA);
        cout << pA << " casts to " << pC << ": Unrelated-cast: Valid for null" << endl;
        pA = &a; void *pV = dynamic_cast<void*>(pA);
        cout << pA << " casts to " << pV << ": Cast-to-void: Valid" << endl;
        // pA = dynamic_cast<A*>(pV); // error: 'void *': invalid expression type for
dynamic_cast
```

Output:

00EFFCA8 casts to 00EFFCA8: Up-cast: Valid
00EFFCA8 casts to 00EFFCA8: Down-cast: Valid
00EFFCB4 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFCB4 casts to 00EFFCB4: Cast-to-void: Valid

Example 2:

```
#include <iostream>
#include <typeinfo>
```

```
using namespace std;
class A { public: virtual ~A() { } };
class B: public A { };
class C { public: virtual ~C() { } };
int main() { A a; B b; C c;
        try { B &rB1 = b;
                A &rA2 = dynamic_cast<A&>(rB1);
                cout << "Up-cast: Valid" << endl;
                A &rA3 = b;
                B &rB4 = dynamic_cast<B&>(rA3);
                cout << "Down-cast: Valid" << endl;
                try { A &rA5 = a;
                        B &rB6 = dynamic_cast<B&>(rA5);
                } catch (bad_cast e) { cout << "Down-cast: Invalid: " << e.what() << endl; }
                try { A &rA7 = (A&)c;
                        C &rC8 = dynamic_cast<C&>(rA7);
                } catch (bad_cast e) { cout << "Unrelated-cast: Invalid: " << e.what() << endl; }
        } catch (bad_cast e) { cout << "Bad-cast: " << e.what() << endl; }
}
```

Output:

Up-cast: Valid
Down-cast: Valid
Down-cast: Invalid: Bad dynamic_cast!
Unrelated-cast: Invalid: Bad dynamic_cast!

- **typeid Operator:**
  • typeid operator is used where the dynamic type of a polymorphic object must be known and for static type identification
  • typeid operator can be applied on a type or an expression
  • typeid operator returns const std::type info. The major members are:
  ◦ operator==, operator!=: checks whether the objects refer to the same type
  ◦ name: implementation-defined name of the type
  • typeid operator works for polymorphic type only (as it uses RTTI – virtual function table)
  • If the polymorphic object is bad, the typeid throws bad typeid exception

Exception Handling In CPP:

- ***Example:***

```
#include <iostream>
#include <exception>
using namespace std;
class MyException: public exception {};
class MyClass {
   public: ~MyClass() {}
};
void h() {
   MyClass h_a;
   //throw 1; // Line 1
   //throw 2.5; // Line 2
```

```
    //throw MyException(); // Line 3
    //throw exception(); // Line 4
    //throw MyClass(); // Line 5
} // Stack unwind, h_a.~MyClass() called
// Passes on all exceptions
void g() {
    MyClass g_a;
    try {
        h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1
    catch (int) {
        cout << "int\n";
    }
    // Catches exception from Line 2
    catch (double) {
        cout << "double\n";
    }
    // Catches exception from Line 3-5 & passes on
    catch (...) {
        throw;
    }
} // Stack unwind, g_a.~MyClass() called

void f() {
    MyClass f_a;
    try {
        g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException) {
        cout << "MyException\n";
    }
    // Catches exception from Line 4
    catch (exception) {
        cout << "exception\n";
    }
    // Catches exception from Line 5 & passes on
    catch (...) {
        throw;
    }
} // Stack unwind, f_a.~MyClass() called
int main() {
    try {
        f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...) {
```

```
        cout << "Unknown\n";
    }
    cout << "End of main()\n";
}
```

Note: catch(...) block must be the last catch block because it catches all exceptions
If Base Class catch block precedes Derived Class catch block
◦ Compiler issues a warning and continues
◦ Unreachable code (derived class handler) ignored

- **(re)-throw:**
  ◦ catch may pass on the exception after handling
  ◦ Re-throw is not same as throwing again!
  Example:

```
try {
    ...
} catch (Exception & ex) {
    // Handle and
    ...
    // Pass-on
    // throw ex; throws again
    throw; //re - throw
    // No copy
    // No Destruction
}
```

- **Exceptions in std::exception:**
  • logic error: Faulty logic like violating logical preconditions or class invariants (may be preventable)
  ◦ invalid argument: An argument value has not been accepted
  ◦ domain error: Situations where the inputs are outside of the domain for an operation
  ◦ length error: Exceeding implementation defined length limits for some object
  ◦ out of range: Attempt to access elements out of defined range
  • runtime error: Due to events beyond the scope of the program and can not be easily predicted
  ◦ range error: Result cannot be represented by the destination type
  ◦ overflow error: Arithmetic overflow errors (Result is too large for the destination type)
  ◦ underflow error: Arithmetic underflow errors (Result is a subnormal floating-point value)
  • bad typeid: Exception thrown on typeid of null pointer
  • bad cast: Exception thrown on failure to dynamic cast
  • bad alloc: Exception thrown on failure allocating memory
  • bad exception: Exception thrown by unexpected handler

Templates:

//• In ”C-String Comparison” – swapping parameters changes the result – actually compares pointers (pitfall of macros)

- **Function Template:**
  Example:

```
#include <iostream>
using namespace std;
template < class T >
    T Max(T x, T y) {
```

```
    return x > y ? x : y;
  }
template < > // Template specialization for 'char *' type if not provided compares address
  char * Max < char * > (char * x, char * y) {
    return strcmp(x, y) > 0 ? x : y;
  }
int main() {
  int a = 3, b = 5, iMax;
  double c = 2.1, d = 3.7, dMax;
  iMax = Max < int > (a, b);
  cout << "Max(" << a << ", " << b << ") = " << iMax << endl; // Output: Max(3, 5) = 5
  dMax = Max < double > (c, d);
  cout << "Max(" << c << ", " << d << ") = " << dMax << endl; // Output: Max(2.1, 3.7) = 3.7

  char * s1 = new char[6], * s2 = new char[6];
  strcpy(s1, "black");
  strcpy(s2, "white");
  cout << "Max(" << s1 << ", " << s2 << ") = " << Max < char * > (s1, s2) << endl;
  // Output: Max(black, white) = white
  strcpy(s1, "white");
  strcpy(s2, "black");
  cout << "Max(" << s1 << ", " << s2 << ") = " << Max < char * > (s1, s2) << endl;
  // Output: Max(black, white) = white
}
```

Example:

```
#include <iostream>

using namespace std;
template < class T > T Max(T x, T y) {
  return x > y ? x : y;
}
int main() {
  int a = 3, b = 5, iMax;
  double c = 2.1, d = 3.7, dMax;
  iMax = Max(a, b); // Type 'int' inferred from 'a' and 'b' parameters types
  cout << "Max(" << a << ", " << b << ") = " << iMax << endl;
  // Output: Max(3, 5) = 5
  dMax = Max(c, d); // Type 'double' inferred from 'c' and 'd' parameters types
  cout << "Max(" << c << ", " << d << ") = " << dMax << endl;
  // Output: Max(2.1, 3.7) = 3.7
}
```

Example:

```
#include <iostream>
#include <cmath>
#include <cstring>
using namespace std;
class Complex {
  double re_;
  double im_;
```

```
    public:
      Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) {};
    double norm() const {
      return sqrt(re_ * re_ + im_ * im_);
    }
    friend bool operator > (const Complex & c1, const Complex & c2) {
      return c1.norm() > c2.norm();
    }
    friend ostream & operator << (ostream & os, const Complex & c) {
      os << "(" << c.re_ << ", " << c.im_ << ")";
      return os;
    }
};
template < class T > T Max(T x, T y) {
  return x > y ? x : y;
}
template < > char * Max < char * > (char * x, char * y) {
  return strcmp(x, y) > 0 ? x : y;
}
int main() {
  Complex c1(2.1, 3.2), c2(6.2, 7.2);
  cout << "Max(" << c1 << ", " << c2 << ") = " << Max(c1, c2) << endl;
  // Output: Max((2.1, 3.2), (6.2, 7.2)) = (6.2, 7.2)
}
```

Example:

```
#include <iostream>
#include <cstring>
using namespace std;
template < class T > T Max(T x, T y) {
  return x > y ? x : y;
}
template < > char * Max < char * > (char * x, char * y) // Template specialization
{
  return strcmp(x, y) > 0 ? x : y;
}
template < class T, int size > T Max(T x[size]) { // Overloaded template function
  T t = x[0];
  for (int i = 0; i < size; ++i) {
    if (x[i] > t) t = x[i];
  }
  return t;
}
int main() {
  int arr[] = {
    2, 5, 6, 3, 7, 9,4
  };
  cout << "Max(arr) = " << Max < int, 7 > (arr) << endl; // Output: Max(arr) = 9
}
```

- typename:
  • Consider:

```
template <class T> f (T x) {
        T::name * p;
}
```

• What does it mean?
◦ T::name is a type and p is a pointer to that type
◦ T::name and p are variables and this is a multiplication
• To resolve, we use keyword typename:

```
template <class T> f (T x) { T::name * p; } // Multiplication
template <class T> f (T x) { typename T::name * p; } // Type
```

• The keywords class and typename have almost the same meaning in a template parameter
• typename is also used to tell the compiler that an expression is a type expression

- **Template Classes:**

Example: //Stack.h

```
template<class T>
class Stack {
  T data_[100];
  int top_;
public:
  Stack() :top_(-1) { }
  ~Stack() { }
  void push(const T& item) { data_[++top_] = item; }
  void pop() { --top_; }
  const T& top() const { return data_[top_]; }
  bool empty() const { return top_ == -1; }
};

#include <iostream>
#include <cstring>
using namespace std;
#include "Stack.h"

int main() {
  char str[10] = "ABCDE";
  Stack<char> s; // Instantiated for char
  for (unsigned int i = 0; i < strlen(str); ++i)
    s.push(str[i]);
  cout << "Reversed String: ";
  while (!s.empty()) {
    cout << s.top();
    s.pop();
  }
  return 0;
}
```

Example:

```
#include <iostream>
#include <string>
#include <cstring>
```

```
template<class T1 = int, class T2 = std::string> // Version 1 with default parameters
class Student {
   T1 roll_;
   T2 name_;
public:
   Student(T1 r, T2 n) : roll_(r), name_(n) { }
   void Print() const {
      std::cout << "Version 1: (" << name_ << ", " << roll_ << ")" << std::endl;
   }
};

template<class T1> // Version 2: Partial Template Specialization
class Student<T1, char *> {
   T1 roll_;
   char *name_;
public:
   Student(T1 r, char *n) : roll_(r), name_(std::strcpy(new char[std::strlen(n) + 1], n)) { }
   void Print() const {
      std::cout << "Version 2: (" << name_ << ", " << roll_ << ")" << std::endl;
   }
};

int main() {
   Student<int, std::string> s1(2, "Ramesh"); // Version 1: T1 = int, T2 = string
   s1.Print();
   Student<int> s2(11, "Shampa"); // Version 1: T1 = int, default T2 = string
   s2.Print();
   Student<> s3(7, "Gagan"); // Version 1: default T1 = int, default T2 = string
   s3.Print();
   Student<std::string> s4("X9", "Lalita"); // Version 1: T1 = string, default T2 = string
   s4.Print();
   Student<int, char*> s5(3, "Gouri"); // Version 2: T1 = int, T2 = char*
   s5.Print();

   return 0;
}
```

Output:

```
Version 1: (Ramesh, 2)
Version 1: (Shampa, 11)
Version 1: (Gagan, 7)
Version 1: (Lalita, X9)
Version 2: (Gouri, 3)
```

Time pass: A callable entity such as a macro or a function is known as a functor or function object

Function Pointer In C:

• Define a Function Pointer

```
int (*pt2Function) (int, char, char);
```

• Calling Convention

int DoIt (int a, char b, char c); // __cdecl, __stdcall used in MSVC

int DoIt (int a, char b, char c) {

printf ("DoIt\n");

return a+b+c;

}

• Assign Address to a Function Pointer

pt2Function = &DoIt; // OR

pt2Function = DoIt;

• Compare Function Pointers

if (pt2Function == &DoIt) {

printf ("pointer points to DoIt\n");

}

• Call the Function pointed by the Function Pointer

int result = (*pt2Function) (12, 'a', 'b');

- **Using typedef:**
  Example:

```
#include <stdio.h>

typedef int (*pt2Function) (int, char, char);

int DoIt (int a, char b, char c);

int main() {
   pt2Function f = &DoIt; // DoIt
   int result = f(12, 'a', 'b');
   printf("%d", result);
   return 0;
}

int DoIt (int a, char b, char c) {
   printf ("DoIt\n");
   return a + b + c;
}
```

  Output:

```
DoIt
207
```

Function Reference In CPP:

• Define a Function Pointer

int (A::*pt2Member)(float, char, char);

• Calling Convention

class A {

int DoIt (float a, char b, char c) {

cout << "A::DoIt" << endl; return a+b+c; }

};

• Assign Address to a Function Pointer

pt2Member = &A::DoIt;

• Compare Function Pointers

if (pt2Member == &A::DoIt) {

cout <<"pointer points to A::DoIt" << endl;

}

• Call the Function pointed by the Function Pointer

int result = (*this.*pt2Member)(12, 'a', 'b');

- ***Example:***

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

// The four arithmetic operations
float Plus(float a, float b) { return a + b; }
float Minus(float a, float b) { return a - b; }
float Multiply(float a, float b) { return a * b; }
float Divide(float a, float b) { return a / b; }

// Solution with Function pointer
void Switch(float a, float b, float (*pt2Func)(float, float)) {
   float result = pt2Func(a, b);
   cout << "Result := " << result << endl;
}

int main() {
   float a = 10.5, b = 2.5;
   Switch(a, b, &Plus);
   Switch(a, b, &Minus);
   Switch(a, b, &Multiply);
   Switch(a, b, &Divide);
   return 0;
}
```

- ***Example:***

```
int AdderFunction(int a, int b) { // A function
   return a + b;
}
```

```
class AdderFunctor {
public:
   int operator()(int a, int b) { // A functor overrides () operator
      return a + b;
   }
};

int main() {
   int x = 5;
   int y = 7;

   int z = AdderFunction(x, y); // Function invocation

   AdderFunctor aF;
   int w = aF(x, y); // aF.operator()(x, y); -- Functor invocation
}
```

Lambda Function:

- (see insert() and erase() before this in map, vector, list, set)
- ***Example:***

```
sort(vr.begin(), vr.end(),
   [] (const Record& a, const Record& b) // lambda expression as policy [C++11]
   { return a.name < b.name; } // sort by name
);
```


1) Consider the following program.

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

int main(){
    int n = 0x0043;
    printf("%d %o %x %c", n, n, n, n);
    return 0;
}
```

What will be the output/error?

a) 43 43 43 43

b) 67 103 43 C

c) 67 103 43 67

d) Compiler error at LINE-1: type cast from int -> char is invalid

a
b
c
d

3) Consider the following code segment.

```
#include <iostream>
#include <iomanip>

int main () {
    std::cout.precision(4);
    std::cout << std::setfill ('0') << std::setw (8) << (double)10/3;
    return 0;
}
```

What will be the output?

a) 3.333333

b) 3.333000

c) 0003.333

d) 003.3333

a
b
c
d

5) Match the appropriate descriptions about the `fseek` function calls.
Here, `infp` is a file pointer pointing to the beginning of a file in read-write mode.

Function call	Description
1. `fseek(infp, 10, SEEK_SET)`	A. Moves the file pointer to the beginning of the file
2. `fseek(infp, -10, SEEK_CUR)`	B. Moves the file pointer forward from the beginning of the file by 10 positions
3. `fseek(infp, -10, SEEK_END)`	C. Moves the file pointer backwards from the current position in the file by 10 positions
4. `fseek(fp, 0, SEEK_SET)`	D. Moves the file pointer backwards from the end of the file by 10 positions

a) 1-C, 2-A, 3-B, 4-D

b) 1-C, 2-B, 3-A, 4-D

c) 1-B, 2-C, 3-D, 4-A

d) 1-B, 2-D, 3-C, 4-A

a
b
c
d

algorithm Component:

- **r = find(b, e, v)**
  // r points to the first occurrence of v in [b,e)

  r = find_if(b, e, p)
  // r points to the first element x in [b,e) for which p(x)

  x = count(b, e, v)
  // x is the number of occurrences of v in [b,e)

  x = count_if(b, e, p)
  // x is the number of elements in [b,e) for which p(x)

  sort(b, e)
  // sort [b,e) using <

  sort(b, e, p)
  // sort [b,e) using p

  copy(b, e, b2)
  // copy [b,e) to [b2,b2+(e-b))
  // there had better be enough space after b2

  unique_copy(b, e, b2)
  // copy [b,e) to [b2,b2+(e-b))
  // but do not copy adjacent duplicates

  merge(b, e, b2, e2, r)
  // merge two sorted sequence [b2,e2) and [b,e) into [r,r+(e-b)+(e2-b2))

  r = equal_range(b, e, v)
  // r is the subsequence of [b,e) with the value v
  // (basically a binary search for v)

  equal(b, e, b2)
  // do all elements of [b,e) and [b2,b2+(e-b)) compare equal?

numeric Component:

- **accumulate**
  // Accumulate values in range

  adjacent_difference
  // Compute adjacent difference of range

  inner_product
  // Compute cumulative inner product of range

  partial_sum
  // Compute partial sums of range

iota [C++11]

```
// Store increasing sequence
```

- **accumulate:**

```
#include <iostream>
#include <vector>
#include <numeric> // accumulate
using namespace std;

void f(vector<double>& vd, int* p, int n) {
   double sum = accumulate(vd.begin(), vd.end(), 0.0); // 12.3 : add the elements of vd
   // note: the type of the 3rd argument, the initializer, determines the precision used
   int si = accumulate(p, p + n, 0); // 10 : sum the ints in an int. p+n means (roughly) &p[n]
   long sl = accumulate(p, p + n, long(0)); // 10 : sum the ints in a long
   double s2 = accumulate(p, p + n, 0.0); // 10 : sum the ints in a double
   // popular idiom, use the variable you want the result in as the initializer:
   double ss = 0;
   ss = accumulate(vd.begin(), vd.end(), ss); // 12.3 : do remember the assignment
}

int main() {
   vector<int> v = { 1, 2, 3, 4 };
   int sum = accumulate(v.begin(), v.end(), 0); // 10
   vector<double> vd = { 1.5, 2.7, 3.2, 4.9 };
   f(vd, &v[0], v.size());
}
```

Note: if you want to perform some other operation other than sum you can pass that function as a parameter at last.

```
accumulate(vr.begin(), vr.end(), 0.0, // use a lambda [C++11]
        [](double v, const Record& r) { return v + r.unit_price * r.units; }
);
```

- **inner_product:**

```
#include <iostream>
#include <vector>
#include <numeric> // inner_product
using namespace std;

int main() {
   // calculate the Dow-Jones industrial index:
   // share price for each company
   vector<double> dow_price = { 81.86, 34.69, 54.45 };
   // weight in index for each company
   vector<double> dow_weight = { 5.8549, 2.4808, 3.8940 };
   // multiply (price, weight) pairs and add
   double dj_index = inner_product(
      dow_price.begin(), dow_price.end(), dow_weight.begin(), 0.0);
   cout << dj_index << endl; // 777.369
}
```

Note: by default it is sum of products but you can spectify the operations by passing two functions

C++ 11

- **auto & decltype:**

```
auto v1 = li;                   // v1: std::list<int>
auto& v2 = li;                  // v2: const std::list<int>&
decltype(x) i1;                 // i1's type is int
decltype(ptr) p1;               // p1's type is int*

Parenthesis can matter

struct S { double d; };
const S* p;
decltype(p->d) x1;              // double
decltype((p->d)) x2;            // const double&
```

Example:

```
int main() {
   int a = 5; // int
   int& b = a; // int&
   const int c = 7; // const int
   const int& d = c; // const int&

   // auto never deduces adornments like cv-qualifier or reference
   auto a_auto = a; // int
   auto b_auto = b; // int
   auto c_auto = c; // int
   auto d_auto = d; // int

   // cv-qualifier or reference needs to be explicitly added
   auto& b_auto_ref = a; // int&
   const auto c_auto_const = a; // const int

   // decltype deduces the complete type of the expression
   decltype(a) a_dt; // int // [C++14] decltype(auto) a_dt_auto = a; // int
   decltype(b) b_dt = b; // int& // [C++14] decltype(auto) b_dt_auto = b; // int&
   decltype(c) c_dt = c; // const int // [C++14] decltype(auto) c_dt_auto = c; // const int
   decltype(d) d_dt = d; // const int& // [C++14] decltype(auto) d_dt_auto = d; // const int&
}
```

- We really need decltype if we need a type for something that is not a variable, such as a return type.

Example:

```
template<class T, class U>
decltype(x*y) mul(T x, U y) { return x*y; } // scope problem! types of x and y not known

template<class T, class U>
(decltype*(T*)(0)**(U*)(0)) mul(T x, U y) { return x*y; } // ugly! and error prone

template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
(or)
template<class T, class U>
```

```
auto mul(T x, U y) { return x*y; }
(or)
template<class T, class U>
decltype(auto) mul(T x, U y) { return x*y; }
```

- **Diff b/w auto and decltype:**

```
#include <iostream>

// returns prvalue: plain auto never deduces to a reference. prvalue is a pure rvalue – TBD later
template<typename T>
auto foo(T& t) { return t.value(); }

// return lvalue: auto& always deduces to a reference
template<typename T>
auto& bar(T& t) { return t.value(); }

// return prvalue if t.value() is an rvalue
// return lvalue if t.value() is an lvalue
// decltype(auto) has decltype semantics (without having to repeat the expression)
template<typename T>
decltype(auto) foobar(T& t) { return t.value(); }

int main() {
   struct A { int i = 0 ; int& value() { return i ; } } a;
   struct B { int i = 0 ; int value() { return i ; } } b;

   foo(a) = 20; // *** error: expression evaluates to prvalue of type int
   foo(b); // fine: expression evaluates to prvalue of type int

   bar(a) = 20; // fine: expression evaluates to lvalue of type int&
   bar(b); // *** error: auto& always deduces to a reference (int&) - bar(b) needs an initializer

   foobar(a) = 20; // fine: expression evaluates to lvalue of type int&
   foobar(b); // fine: expression evaluates to prvalue of type int
}
```


- **Initializer Lists:**

• auto deduces std::initializer list for braced initializers:

```
auto i = { 2, 4, 6, 8 }; // i is std::initializer_list<int>
```

• In general, templates deduce no type for braced initializers:

```
template<typename T> void f(T param) { ... }
f({ 2, 4, 6, 8}); // error! no type deduced for { 2, 4, 6, 8 }
```

• Especially for single-element braced initializers, this can confuse:

```
auto i1 = 10; // i1 is int
auto i2(10); // i2 is int
auto i3 {10}; // i3 is std::initializer_list<int>
```

• Particularly when such variables interact with overload resolution:

```
std::vector<int> v1(i1); // v1.size() == 10, values == 0
std::vector<int> v2(i2); // v1.size() == 10, values == 0
std::vector<int> v3(i3); // v1.size() == 1, value == 10
```

• Use care when initializing auto variables with braced initializers!

- Member and heap arrays are impossible:

```
class Widget {
        public: Widget(): data(???) {}
        private: const int data[5]; // not initializable
};
const float * pData = new const float[4]; // not initializable

// Brace-initialized variables may use =:
const int val1 = {5};
const int val2 = {5};
int a[] = { 1, 2, val1, val1 + val2 };
struct Point1 { ... };
const Point1 p1 = {10, 20};
class Point2 { ... };
const Point2 p2 = {10, 20};
const std::vector<int> cv = { a[0], 20, val2 };

// Other uses of brace initialization cannot:
class Widget { // Not allowed in member initialization
public:
  Widget(): data = {1, 2, a[3], 4, 5} {} // error!
private:
  const int data[5];
};

// Not allowed in dynamic allocation
const float *pData = new const float[4] = { 1.5, val1 - val2, 3.5, 4.5 }; // error!

// Not allowed in return
Point2 makePoint() { return = { 0, 0 }; } // error!

// Not allowed in function parameters
void f(const std::vector<int>& v);
f( = { val1, val2, 10, 20, 30 });
```

• And T var = expr syntax cannot call explicit constructors:

```
class Widget {
public:
explicit Widget(int);
...
};
Widget w1(10); // okay, direct init: explicit ctor callable
Widget w2{10}; // okay, direct init: explicit ctor callable
Widget w3 = 10; // error! copy init: explicit ctor not callable
Widget w4 = {10}; // error! copy init: explicit ctor not callable
```

• Develop the habit of using brace initialization without =

```
// Sole exception: implicit narrowing
// C++03 allows it via brace initialization, C++11 does not
struct Point { int x, y; };
```

```
Point p1 = { 1, 2.5 }; // fine in C++03
// implicit double => int conversion
// error in C++11
Point p2 = { 1, static_cast<int>(2.5) }; // fine in both C++03 and C++11

// Direct constructor calls and brace initialization thus differ subtly:
class Widget {
public:
   Widget(unsigned u);
   // ...
};

int i;
// ...
Widget w1(i); // okay, implicit int => unsigned
Widget w2{i}; // error! int => unsigned narrows
unsigned u;
Widget w3(u); // fine
Widget w4{u}; // also fine, same as w3's init.
```

- **constexpr:**

Example:

```
enum Flags { good = 0, fail = 1, bad = 2, eof = 4 };

constexpr int operator|(Flags f1, Flags f2) { //a function can always have constexpr in most of the cases
   return Flags(int(f1) | int(f2));
}

void f(Flags x) {
   switch (x) {
      case bad: /* ... */ break;
      case eof: /* ... */ break;
      case bad | eof: /* ... */ break;
      default: /* ... */ break;
   }
}
```

• Here constexpr says that the function must be of a simple form so that it can be evaluated at compile time if given constant expressions arguments
• In addition to be able to evaluate expressions at compile time, we want to be able to require expressions to be evaluated at compile time
• constexpr in front of a variable definition does that (and implies const):

Example:

```
constexpr int x1 = bad | eof; // okay

void f(Flags f3) {
   constexpr int x2 = bad | f3; // error: cannot evaluate at compile time (as f3 is passed at run time)
```

```
    int x3 = bad | f3; // okay
}
```

• Typically we want the compile-time evaluation guarantee for global or namespace objects, often for objects we want to place in read-only storage
• This also works for objects for which the constructors are simple enough to be constexpr and expressions involving such objects:

Example:

```
struct Point {
    int x, y;
    constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};

constexpr Point origo(0,0); // to become a constexpr one must refer to a constexpr or a constant. Note that nothing should take place at runtime…..
constexpr int z = origo.x;
constexpr Point a[] = { Point(0,0), Point(1,1), Point(2,2) };
constexpr int x = a[1].x; // x becomes 1
```

Note that the constructor can still be used in the usual way with non-constant parameters too

◦ const's primary function
. is to express the idea that an object is not modified through an interface (even though the object may very well be modified through other interfaces)
. It just so happens that declaring an object const provides excellent optimization opportunities for the compiler
. In particular, if an object is declared const and its address is not taken, a compiler is often able to evaluate its initializer at compile time (though that's not guaranteed) and keep that object in its tables rather than emitting it into the generated code
◦ constexpr's primary function
. is to extend the range of what can be computed at compile time, making such computation type safe and also usable in compile-time contexts (such as to initialize enumerator or integral template parameters)
. Objects declared constexpr have their initializer evaluated at compile time
. they are basically values kept in the compiler's tables and only emitted into the generated code if needed

• constexpr needs compile-time constant for initialization whereas const treats the initialized value as constant in run-time

Example:

```
#include <iostream>

constexpr int m = 100; // Okay: m is 100: compile-time constant

// const will also work
// if something depends on the parameter passed to a function and the parameter is not a constexpr then there results a error if you want to make that something constexpr
```

```
void f(int n) {
   constexpr int c1 = m + 1; // Okay: c1 is 101: compile-time constant
   // constexpr int c2 = n + 1; // Error: n is not compile-time constant
   const int c2 = n + 1; // Okay: but value of c2 cannot be changed
   // constexpr int c3 = c2 + 1; // Error: c2 is not compile-time constant
   const int c3 = c2 + 1; // Okay: but value of c3 cannot be changed
   std::cout << c1 << ' ' << c2 << ' ' << c3 << std::endl; // 101 11 12
}

int main() {
   f(10);
}
```

• constexpr cannot be used for all functions. For example:
constexpr int add_vectors_size(const vector<int>& a, const vector<int>& b)
{ return a.size() + b.size(); } // a.size() is not compile time constant
gives compilation error on a.size() as size of a vector is not compile time constant
• However, the following works fine:

Example:

```
#include <iostream>
#include <array> // Fixed size array
using namespace std;

template<size_t N1, size_t N2>
constexpr int add_arrays_size(const array<int, N1>& a, const array<int, N2>& b)
{
   return a.size() + b.size(); // a.size() is compile time constant
} // you can return but you cant assign it to constexpr

int main() {
   array<int, 10> p;
   array<int, 20> q;

   static constexpr auto n = add_arrays_size(p, q);
   cout << n << endl; // 30
}
```


Consider the following code segment.

```
#include <iostream>

class data{
    public:
        constexpr data(int n = 0) : n_{n} { }
    private:
        int n_;
};

const int num_gen1(){
    return 1;
}

constexpr int num_gen2(const int i){
    return i + 10;
}

int main(){
    int i = 10;
    const int j = 20;
    constexpr data d1(1);              //LINE-1
    constexpr data d2(i);              //LINE-2
    constexpr data d3(num_gen1());     //LINE-3
    constexpr data d4(num_gen2(j));    //LINE-4
    return 0;
}
```

Thing that are clear on constexpr

1) You can't assign something that isn't const to a constexpt in any function
2) You cant assign something that depends on function parameter to constexpr
3) Only constexpr functions can be assigned to constexpr variables(what they return)
4) You cant assign a const function return value to a constexpr

Consider the following code segment (in C++11).

```
#include <iostream>

void show(int* ip){ /* code */ }

template<typename Func, typename Param>
void call(Func fn, Param p){
    fn(p);
}

int main(){
    int i = 10;
    call(show, &i);          //LINE-1
    call(show, i);           //LINE-2
    call(show, NULL);        //LINE-3
    call(show, nullptr);     //LINE-4
    return 0;
}
```

Choose the call/s to call function that will result in compiler error/s.

a) LINE-1

b) LINE-2

c) LINE-3

d) LINE-4

1 - Line 2 and 3

Consider the following code segment.

```
#include <iostream>

void print(char* str){ /*some code*/ }

template<typename Fun, typename Param>
void caller(Fun func, Param p){
    func(p);
}

int main(){
    char s[2] = "0";
    caller(print, s);          //LINE-1
    caller(print, 0);          //LINE-2
    caller(print, s[1]);       //LINE-3
    caller(print, nullptr);    //LINE-4
    return 0;
}
```

Which of the following lines generate/s compiler error?

a) LINE-1

b) LINE-2

c) LINE-3

d) LINE-4

2 Line 2 and 3

- **noexcept:**

```
// Not prepared to handle memory exhaustion
vector<double> my_computation(const vector<double>& v) noexcept {
   vector<double> res(v.size()); // might throw
   for(int i; i<v.size(); ++i)
      res[i] = sqrt(v[i]);
   return res;
}

//if throws error then terminate program
```

- **nullptr:**
  - nullptr is a literal denoting the null pointer
    - Literal of type std::nullptr t in <cstddef>
    - Convertible to any pointer type and to bool, but nothing else
    - It is not an integer and cannot be used as an integral value

  NULL or 0 causes confusion in following cases that nullptr can resolve:
    - Function Overload Resolution
    - Forwarding Templates

Example:

```
int* q = nullptr; // q is null
char* p = nullptr; // p is null
char* p1 = 0; // 0 still works, p1 is null and p == p1
char* p2 = NULL; // p2 is null
if (p) ... // compiles but fails
if (p == p1) ... // compiles and succeeds
if (q == p2) ... // error: comparison between distinct pointer types int* and char*
void g(int);
g(nullptr); // error: nullptr is not an int. cannot convert std::nullptr_t to int
int i = nullptr; // error: nullptr is not an int. cannot convert std::nullptr_t to int
void f(int); void f(int*); // Function overload resolution
f(0); // call f(int)
f(nullptr); // call f(int*)
f(NULL); // error: call of overloaded f(NULL) is ambiguous for f(int) and f(int*)
void h(int*); // h(0) and h(nullptr) are okay
template<typename F, typename P> // Forwarding template
void logAndCall(F func, P param) {
  func(param); // make log entry ..., then invoke func on param
}
logAndCall(h, 0); // error: P deduced as int, and h(int) invalid
logAndCall(h, NULL); // error: P deduced as long int, and h(long int) invalid
logAndCall(h, nullptr); // P deduced as std::nullptr_t, and h(std::nullptr_t) is okay
```

- **inline namespaces:**

Example:

```
// C++11: inline namespaces
#include <iostream>
using namespace std;

// inline namespace ns1; for global access
namespace ns1 {
  int v1 = 2;
  inline namespace ns2 {
    int v2 = 3;
    inline namespace ns3 {
      int v3 = 5;
    }
  }
}
int main() {
  // Qualified by enclosing namespace
  cout << ns1::v1 << ' ';
  cout << ns1::v2 << ' ';
  cout << ns1::ns2::v3 << ' ';
  cout << ns1::v3 << endl;
}
```

Output:
2 3 5 5

- **static_assert:**
  • A static (compile time) assertion consists of a constant expression and a string literal:
  static_assert(expression, string);
  • The compiler evaluates the expression and writes the string as an error message if the expression is false

- **User defined literals:**
  Example:

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

std::string operator"" _s(const char* p, size_t n) { // std::string literal
   return string(p, n); // requires free store allocation
}

template<class T>
void f(const T& a) {
   cout << a << endl;
}

int main() {
   f("Hello");     // pass pointer to char* => const char (&)[6]
   f("Hello"_s);   // pass (5-character) std::string object
   f("Hello\n"_s);  // pass (6-character) std::string object
}
```


• Digit Separator

◦ In C++14, the single-quote character ’ can be used anywhere within a numeric literal for aesthetic readability. It does not affect the numeric value
auto million = 1’000’000;
auto pi = 3.14159’26535’89793;

• Binary Literals

◦ C++14 supports binary literals:
auto a1 = 42; // ... decimal
auto a2 = 0x2A; // ... hexadecimal
auto a3 = 0b101010; // ... binary
◦ This works well in combination with the new ’ digit separators, for example, to separate nybbles or bytes:
auto a = 0b100’0001; // ASCII ’A’

Universal References:

- C++11's rules take rvalue references into account:
  ◦ T& & => T& // from C++03
  ◦ T&& & => T& // new for C++11
  ◦ T& && => T& // new for C++11
  ◦ T&& && => T&& // new for C++11

  • Summary:
  ◦ Reference collapsing involving a & is always T&
  ◦ Reference collapsing involving only && is T&&

  • Function templates with a T&& parameter need not generate functions taking a T&& parameter!
  template<typename T> void f(T&& param); // note non-const rvalue reference
  • T's deduced type depends on what is passed to param:
  ◦ Lvalue ⇒ T is an lvalue reference (T&)
  ◦ Rvalue ⇒ T is a non-reference (T)

  • In conjunction with reference collapsing:

  int x;
  f(x); // lvalue: generates f<int&>(int& &&), calls f(int&)
  f(10); // rvalue: generates f<int>(int&&), calls f(int&&)
  TVec vt; // typedef vector<int> TVec;
  // TVec createTVec();

  f(vt); // lvalue: generates f<TVec&>(TVec& &&), calls f(TVec&)
  f(createTVec()); // rvalue: generates f<TVec>(TVec&&), calls f(TVec&&)

  • T&& really is a magical reference type!
  ◦ For lvalue arguments, T&& becomes T& => lvalues can bind
  ◦ For rvalue arguments, T&& remains T&& => rvalues can bind
  ◦ For const/volatile arguments, const/volatile becomes part of T
  ◦ T&& parameters can bind anything

  • Two conceptual meanings for T&& syntax:
  ◦ Rvalue reference. Binds rvalues only
  void f(Widget&& param); // takes only non-const rvalue
  ◦ Universal reference. Binds lvalues and rvalues
  template<typename T>
  void f(T&& param); // takes lvalue or rvalue, const or non-const
  . Really an rvalues reference in a reference-collapsing context

  • auto type deduction ≡ template type deduction, so auto&& variables are also universal references:

```
int calcVal();
int x;
auto&& v1 = calcVal(); // deduce type from rvalue => v1's type is int&&
auto&& v2 = x; // deduce type from lvalue => v2's type is int&
```

• Note that decltype()&& does not behave like a universal references as it does not use template type deduction:

```
decltype(calcVal()) v3; // deduced type is int
decltype(x) v4; // deduced type is int
```

- **Perfect Forwarding:**
  • What is Perfect Forwarding?
  ◦ Perfect forwarding allows a template function that accepts a set of arguments to forward these arguments to another function whilst retaining the lvalue or rvalue nature of the original function arguments.
  ----------broken----------


```
#include <iostream>

class Data {
   int i;
public:
   Data(): i(0) { } // a UDT
};
// binds with lvalue parameter
void g(const int&) {
   std::cout << "int& in g" << "; ";
}
// binds with rvalue parameter
void g(int&&) {
   std::cout << "int&& in g" << "; ";
```

```
}
// binds with lvalue parameter
void h(const Data&) {
   std::cout << "Data& in h" << std::endl;
}
// binds with rvalue parameter
void h(Data&&) {
   std::cout << "Data&& in h" << std::endl;
}
template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) { // universal ref. gets lvalue or rvalue from arg by template type deduction
   g(p1); // always binds with lvalue parameter as p1 is an lvalue in f
   h(p2); // always binds with lvalue parameter as p2 is an lvalue in f
}
int main() {
   int i { 0 };
   Data d;
   // (lvalue, lvalue) binds with int& in g; Data& in h
   f(i, d);
   // (rvalue, lvalue) binds with int& in g; Data& in h
   f(std::move(i), d);
   // (lvalue, rvalue) binds with int& in g; Data& in h
   f(i, std::move(d));
   // (rvalue, rvalue) binds with int& in g; Data& in h
   f(std::move(i), std::move(d));
}
```

----------fixed----------

```
#include <iostream>
class Data {
   int i;
```

```
public:
   Data(): i(0) { } // a UDT
};
// binds with lvalue parameter
void g(const int&) {
   std::cout << "int& in g" << "; ";
}
// binds with rvalue parameter
void g(int&&) {
   std::cout << "int&& in g" << "; ";
}
// binds with lvalue parameter
void h(const Data&) {
   std::cout << "Data& in h" << std::endl;
}
// binds with rvalue parameter
void h(Data&&) {
   std::cout << "Data&& in h" << std::endl;
}
template<typename T1, typename T2>
void f(T1&& p1, T2&& p2) { // universal ref. gets lvalue or rvalue from arg by template type deduction
   g(std::forward<T1>(p1)); // std::forward forwards lvalue arg to lvalue param and
   h(std::forward<T2>(p2)); // rvalue arg to rvalue param
}
int main() {
   int i { 0 };
   Data d;
   // (lvalue, lvalue) binds with int& in g; Data& in h
   f(i, d);
```

```
    // (rvalue, lvalue) binds with int&& in g; Data& in h
    f(std::move(i), d);
    // (lvalue, rvalue) binds with int& in g; Data&& in h
    f(i, std::move(d));
    // (rvalue, rvalue) binds with int&& in g; Data&& in h
    f(std::move(i), std::move(d));
}
```