(5) * `char *str = "myStr";` // read only string and stored in stored read only loc / / Data segment

Here "myStr" is read only but pointer str can be modified.

String array using the 2D array

(I) `char arr[5][10] = { "Rajkalash", "Rajkanay", "Sita", "Gheeta", "Anshu" };`

(II) String array using array of pointer to string

`char *ostr = { "Rajkalash", "Rajkanay", "Sita", "Gheeta", "Anshu" };`

Access 2D array of char using ptr to array

(I) `char arr[5][10] = { "Rajkalash", "Rajkanay", "Sita", "Gheeta", "Anshu" };`

`char *ptrToString[5] = NULL;`

`ptrToString = arr;`

```
for(int i=0; i<5; i++)
    printf("%s", ptrToString[i]);
```

(II) Using ptr to 2D array

`char (*ptr)[5][10] = NULL;`

`ptr = &arr;`

`printf("%s", (*ptr)[i])` to access the string.

(52)

Ex-3 char *arr[5] = { "Rajkumar", "Rajkanya", "Sita", "Geeta", "Sumitra" };

using pointer to 1D array

```
char *(*ptr)[5] = NULL;
```

ptr = &arr

```
printf("%s", (*ptr)[i]); //to access ele
```

element

pointer to pointer

```
char *arr[5] = { "Raj", "Arsh", "Sheel", "Sita",
                 "Sumitra" };
```

```
char **ptr = NULL;
ptr = arr;
printf("%s", ptr[i]);
```

Constant and manifest const

manifest const

- they are defined by #define
- replaced by cpp preprocessor

```
#include <iostream>
#include <cmath>
using namespace
#define TWO 2.
```

constant

- value of const can not be changed after definition

```
const int a = 10;
a = 5 // compiler error

int m
int *p = m
*p = 7
```

p = &m; is a compilation error as m can be changed by *p = 5

Note: A const variable must be initialized when defined. since it contains some garbage value and A-T. depn we can not change later

A variable of any type can be declared as const.

```
typed struct Complex
{
    double re;
    double img
} Complex;
const Complex c = { 2.3, 4.3 };
```

c.re = 5 ✗ error

(54)

~~NULL~~ is manifest constant

manifest const	const variable
(i) Not type safe	type safe
replaced textually by cpp	visible to compiler
not be watched by debugger	watched by debugger
evaluated as many times as replaced	~~E~~valuated only on initialization
file scope	~~F~~ollow scope rule.

const

(I) pointer to const data : where pointee (pointed data) can not be changed.

(II) constant pointer : where pointer can not be change

eg.

```
int m=4;
const int n=5     // Here n is constant
                  // i.e n=6 X
const int *p=&n
```

p pointer to constant data

so *p=6 illegal. but p itself is not constant so p=&m is valid.

ex: (2)

```
int n=5
~~const int * p=&n~~
n=6
```

pointer to const data but data is not constant

It tells compiler that if you go through me it will not

(55) allow you to change.

n = 6 // valid

*p = 6 // Error

ex-3 const int n = 4; → invalid

int *p = &n // Invalid since n is constant. then why we need to change

(12)

we cannot create a non constant pointer to constant variable.

ex-4 int m=4, n=5;

int * const p = &n // Here pointer is constant But pointed data is not

so p = &m // error

*p = 10 // valid.

n = 6 // valid

We can create a constant pointer to non constant data. ~~But the reverse~~ But. can not change the data using *p. and reverse is not true.

(5) const int m = 4

const int * const p = &m

m = 5 // error

p = &n // error

(56) strdup - returns pointer to newly terminated byte string which is duplicate of string pointed by s; it gives memory allocated through malloc.

Ex. char source[] = "LLT";

char *str = strdup(source);

Ex.1 char *str = strdup("LLT Kharagpur");

str[0] = 'N';

str = strdup("LLT madras");

To stop editing the Name:

```
const char *str = strdup("LLT Kharagpur");
str[0] = 'N' // error
```

To stop changing the Name:

```
char * const str = strdup("LLT Kharagpur");
str = strdup("NLT"); // error
```

To stop both

```
const char * const str = strdup("LLT Kharagpur");
str[0] = 'N'; // error
str = strdup("NLT"); // error
```

(57)

volatile: The value of volatile variable may be different everytime it is read. Even if no assignment has been made to it.

- a variable is taken as volatile if it can be changed by hardware like kernel, the thread etc.

NIW port

ex: `static int i;`

```
void fun(void)
{   i=0;
    while(i!=100)
}
```

infinite loop — compiler will optimize like

```
static int i;
void fun(void)
{   i=0;          // compiler optimizes
    while(1);
}
```

`static volatile int i;`

```
void fun(void)
{   i=0;          // compiler does not optimize
    while(i!=100)
}
```

It ensures that at any point i value will change by kernel or some other program and it will becomes 100 and it will exit the loop.

in_line function.

macro with parameter — simple text based expansion

```
#include <iostream>
using namespace std;
#define SQUARE(x) x*x
int main()
{
    int a=3, b;
    b= SQUARE(a);
    cout << b << endl;
    return 0;
}
```

b = a*a
= 3*3
= 9

when b = SQUARE(a+1)

O/P: a+1 * a+1 expected is 16

= 3+1*3+1
= 7

To fix this

```
#define SQUARE(x) (x)*(x)
```

Now b = SQUARE(a+1)

b = (a+1)(a+1)
= (3+1)(3+1)
= 4*4 = 16

Now b = SQUARE(++a)

expected = 16 But

b = (++a) * (++a)

= 5 * 5 = 25

There is no easy fix for this

inline function

- just a normal function
- fn name preceded by keyword inline.
- when the function is called then the actual fn call doesn't happen, but wherever the code fn has compiler put that code at the call side.
- overhead of parameter passing b/n caller and callee is avoided.

Replacing macro with inline

using namespace std;

inline int SQUARE(a) { return a*a; }

b = SQUARE(a);

macro	inline
expanded at the place of calls	expanded at the place of call
Efficient in execution.	efficient in execution
code bloats	code bloats
syntatic and semantic pitfalls.	No pitfalls

(6) Type checking for parameter is not done	Type checking for parameter is robust.
Helps to write macro swap for all types	Needs template for same purpose.
errors are not checked during compilation	errors are checked during compilation
Not available to debugger	Available to debugger in DEBUG build.
	RELEASE build.

Limitation of inline

- It's directive (suggestion) - compiler may not inline fn with large body
- may not be recursive
- fn body is needed for inlining at the time of fn call. Hence implementation hiding is not possible. Hence inline fn should be present in Header file
- inline fn must not have two different definitions

(61) functions with default value parameter

ex: int myfun(int a=0)
{
return a;
}

call
int y = myfun(10);
int y = myfun();

ex-2
int Add(int a=10, int b=20)
{
return (a+b);
}

call int x=5, y=6, z;
z = Add(x,y);
z = Add(x);
z = Add();

* Default argument may be expression as long as they can be computed at compile time

int fun(int = 2+3) Valid

int fun(int = 2+a) Invalid.

* All parameters to the right of a parameter with default argument must have default arguments (Not defaults)

void f(int, double = 0.0, char) → X
Invalid

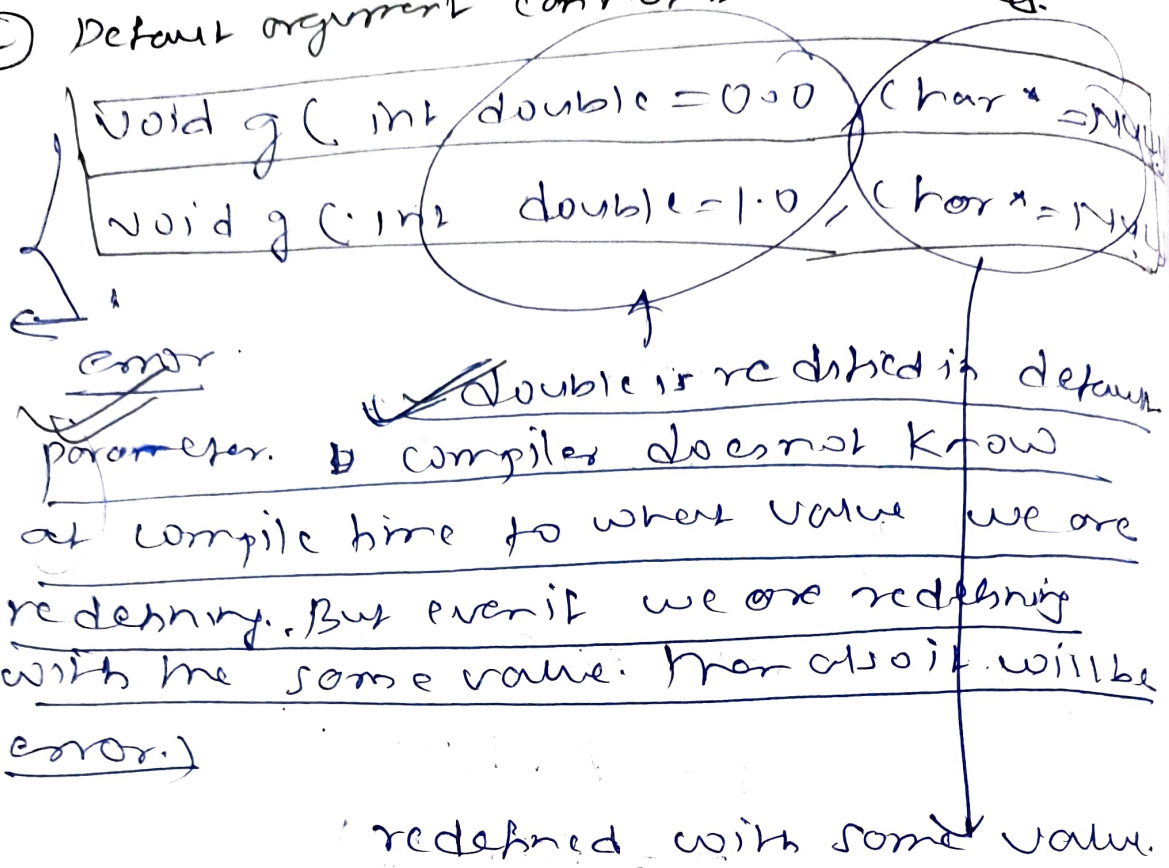
error

(62) void f(int, double = 0.0, char * = NULL)

↑ ↑

all are default

(1) Default argument cannot be redefined.



(3) All non defaulted parameters needed in a call

void g(int, double = 0.0, char * = NULL);

g() // error

(4) parameters are passed from left to right for default value.

g(10, 1.0, NULL) OK by

g(10, 20) // Error

(6) default parameters should be supplied in a header file and not in the definition of a function.

Header file: for myfunc.h

```
void g(int, double, char = a);
```

Application file: Apps.cpp.

```
#include "myfunc.h"

void g(int i, double f=0, char ch)

void g(int i=0 double f char ch)
```

default type and value

OK

default int value. Combing all these fn will be ~~slight~~ ~~slight~~ slightly confusing so. Combine all and default in header file.

Function overloading or static polymorphism

Same function Name doing somewhat related functionality but different in type of parameter or no of parameter

- same fn name maybe used in several definition
- fn with same name must have
  - (I) difference in no of formal parameter and/or
  - (II) diff types of formal parameter

happens binding at compile time

- function selection is based on the no and types of the actual parameter at the place of invocation

- function selection (Overload Resolution) is performed by compiler
- Two fn having the same signatures but diff return type results in compile error. to do to attempt to re-define
- allows static polymorphism

ex: some # of parameter

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

To call

```
int x=10, y=25;
double a=2.45, b=6.25;
int result1 = Add(x,y);
result2 = Add(a,b)
```

Different # of parameter

```
int Area(int a, int b){ return a*b; }
int Area(int a){ return a*a; }
```

To call

```
int area1 = Area(x,y);
int area2 = Area(a);
```

(68)

Two fn having the same signature but diff return type cannot be overloaded.

```
int Area(int a, int b){ return axb; }
double Area(int a, int b){ return axb; }
```

↑ error

Overload Resolution

process by which compiler decides among multiple candidate set of overloaded fn that exists, the compiler has to decide which particular fn it has to bind to

To resolve overloaded functions with one parameter

(i) Identify the set of candidate fn

(ii) from the set of candidate fn, identify the set of ~~candidate~~ viable fn

(iii) select the best viable fn through (order is important)

(a) exact match
(b) promotion
(c) standard type conversion
(d) user defined type conversion

ex:

```
int g(double) ---- f1
void f(); ---- f2
void f(int); ---- f3
double h(void); f4
void f(double, double = 3.4) -- f5
```

void h(int, double) — F7

void f(char, char *) — F8

f(5.6)

(I) candidate - F2, F6, F3, F8

(II) viable - F3, F6

(III) Best viable fn (type double) exact match

F6

Overload Resolution: exact match

(I) lvalue to rvalue conversion

↓ ↓

lvalue of variable

address of variable

a = b;

(II) Array to pointer conversion

def: int arr[10];

void f(int *a)

call: f(arr);

(III) Function to pointer conversion

typedef int (*fp)(int);

void f(int, fp)

int g(int);

call: f(5, g) we would have passed f(5, &g)

(iv) Qualification Conversion (T,)

↓ converting pointer (or ref) to const pointer

Overload resolution fails

```
int fun(float a) { ... }
int fun(float a, int b) { ... }
int fun(float x, int y=10) { ... }
```

```
float p=4.5, t=10.5
int s=30
```

fun(p,s) → matches (2) and (3)

fun(t) → matches (1) and (3)

Results in ambiguity.

Compiler deal with default parameters as a special case of function overloading with some initial value parameter.

ex. int f(int a=1, int b=2); ↓ can be overloaded as

```
int f();
int f(int);
int f(int, int);
```

Note: Even we overloaded fn some parameter may default but as long as we can resolve from the set of prototype fn with the default parameters from the candidate

```
int Area(int a, int b=10)
double Area(double c, double d)

int x=10, y=12, t
double z=20.5  y=5.0, f

t = Area(a); ✓

f = Area(z, y)
```

↑ promoted to double

* Default parameter and fn overloading can be mixed together as long as the whole thing can be resolved.

look at whether they can be actually resolved or not

ex. 2

```
int f();
int f(int =0);
int f(int, int);

int x=5, y=6;

f(); → which one to call
f(x)   ambiguous call
f(x,y)
```

(67)

Operator Overloading

What's difference b/w an operator and a fn?

They do similar kind of functionality. So whenever we use operator we can replace it with function and vice versa

both Operator and ~~Ope~~ fn take some parameter and produces the result.

Operator	Function
(i) Usually written in infix notation a+b, a*b, a/b ++a a?b:c a++	Always written in prefix notation Ex. max(a,b) qsort(int[], int, int, void(*)(void*, void*));
(ii) Operates on one or more operands typically upto 3	(ii) Operates on zero or more arguments
(iii) produces one result	(iii) produces upto one result
(iv) Order of Operation is decided by precedence and associativity a+b*c	(iv) order of appln is decided by depth of nesting f(a, g(b,c), d);
(v) They are predefined.	(v) can be defined as needed.

Operator function in C++

- introduces a new keyword : operator.
- Each operator is associated with an operator function that defines its behaviour

Operator expression	Operator function
a+b	operator + (a, b)
a = b	operator = (a, b)
c = a+b	operator = (c, operator+(a, b))

* operator functions are implicit for predefined operators of built-in types and can not be redefined

- An operator function may have signature as:

```
MyType a, b; // error or should not be from built in type
...
MyType operator+( MyType, MyType); // operator function
```

- a+b → this will call b operator+(a, b)
- C++ allows user to define an operator function and overload it

C++

~~typedef String *~~

typedef struct _String { char *str; } String;

String fName, lName, name;

fName.str = strdup ("Partha");

lName.str = strdup ("Das");

name.str = (char*) (malloc (strlen (fName.str) + strlen (lName.str) + 1));

strcpy (name.str, fName.str);

strcat (name.str, lName.str);

But we can not use + with operator overloading.

String operator + (const String &S1, const String &S2)

{ String S;

S.str = (char*) malloc (strlen (S1.str) + strlen (S2.len) + 1);

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 |
| or operator+ (fname, lName) |

without object ← compiler will see 0 operands type ' String ' it tries to find is any there any operator fn which takes left and right two string variable of type string.)

ex-2   enum E { c0 = 0, c1 = 1, c2 = 2 };

Ea   E a = c1, b = c2;

int x = -1

x = c1 + c2

c1 and c2 will be first converted to int and then add.

wrapping above fn in enum word.

```
E operator+( E const E &a, const E &b)
{
    unsigned int uia = a, uib = b;
    unsigned int t = a+b (uia + uib) % ;
    return (E) t;
}
```

(13) Now E x=c1, y=c2;

~~int z = c1+c2~~

int z = x+y

~~It has to find operator fn which takes left & right E~~ operands

Compiler sees x is of type E and y is also of type E so it tries to find and operator fn which takes (E, E) and makes sum of type E.

Application

(i) It allows to replace complex fn by just a operator fn or symbol

(ii) we can give a new semantic for my type while the underline type may remain continue to a built in type.

Operator Overloading Rules

* No new operator ~~can~~ is (with new symbol) such as **, <>, or &| can be defined for overloading

* Intrinsic property of the overloaded operator cannot be changed like
  (a) preserve arity
  (b) preserve precedence
  (c) preserve associativity.

(24)

Following operator can be overloaded

[] + - * / % & | ~ != +=

-= *= /= %= &= = &= |=

<< >> >= <= == != < > <= >=

&& || ++ --, ->* -> () []

* unary prefix operator use

```
myType & operator++( myType &s1)
```

* unary postfix operator use

```
myType & operator++( myType &s1,
    int)
```

But during calls no need to pass int

* Following operator can not be overloaded.

* Scope Resolution(::) : because it performs a (compile time) scope resolution rather than on exp. evaluation

* dot(.) : It will raise question whether it is for Object reference or overloading

* Ternary(? :)

Op Overloading exp1 ? exp2 : exp3 would not be able to guarantee that only one of exp2 or exp3 was executed

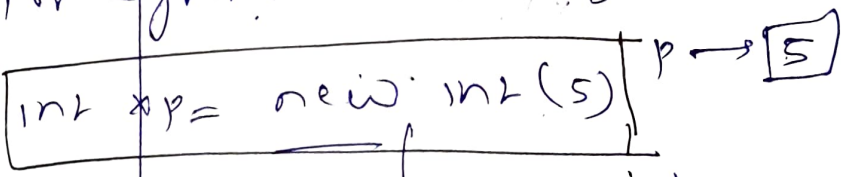
* sizeof : because built in operation such as incrementing a pointer into an array implicitly depends on it.

* The overload of operators && || and (,) loose their special properties: short circuit and sequencing

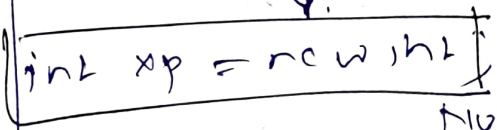
* The overload of Operator -> must either return a raw pointer or return an Object (by reference or by value) for which operator -> is in turn overloaded.

Dynamic memory mgmt in C++

In addition to malloc, calloc, realloc fn C++ introduces new and delete operator for dynamic mem mgmt.



Here we are passing int as datatype to new so. new will create memory for int type and return a pointer of int type



new mem.

no size allocation is needed.
can be initialized, part of C++ core library

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