volatile : modifier that tells the compiler that a variable's value may be changed in ways not explicitly specified by the program

storage class specifiers:

extern = to specify that an object is declared with external linkage elsewhere in the program

static = maintain their values between calls

register = compiler keep the value of a variable in a register of CPU ⇒ much faster speed

Constants : fixed value ⇒ also called literals

---

printf("%c", "%d", "%f", "%s", "%u", "%p", "%%", ...)

%c → char
%d → signed decimal integer
%f → float
%s → string
%u → unsigned decimal int.
%p → pointer
%% → %

scanf("%[]", ---- " " " " " " , &x, &y, &z, ...)

%[] → a set of characters

---

getch() ⇒ reads a character without echo; does not wait for carriage return; not defined by Std. but a common

puts() ⇒ writes a string to the screen

gets() ⇒ reads a string from keyboard

if (condition) { } else { }

switch (X) { case : ' break ; case : ' break ;
else default : ' ' }

&& || !
logical operators :→ to combine simple conditions
(grade == 1) && (age >= 65)

Random number generation:
#include <cstdlib>
srand() → take unsigned integer arguments and seed rand() to produce different sequences of random numbers for each executing program.
rand() → generates unsigned integer between 0 and RAND_MAX.
↑ 32767

Storage classes: auto, register, extern, mutable, static
(determines the period of identifier's existence in memory)
auto storage class → local variables ← global variable
static storage class
storage is allocated and initialized once =0 by default

static = retain their values when the function returns to its caller.
(keeps final value) even when they are out of scope
(static finally)

inline function :- used only with small, frequently used functions.
- before return type to generate a copy of function's code in place (when appropriate) to avoid a function call.

Recursion: $n! = n \times (n-1)!$

$n! = n \cdot (n-1)!$

```
factorial (number)
{ number * factorial(number-1) }
```

```
factorial (n) → number
{   n * factorial(n-1) }
```

Fibonacci Series: 0 1 1 2 3 5 8 13 21 ...

$F(n) = F(n-1) + F(n-2)$

```
fibonacci ( number)
{ if (number == 0 || number == 1)
    { return number; }
  else
    { return fibonacci (number-1) + fibonacci (numb
```

References and reference parameters:

pass-by-reference: the caller gives the called function the ability to access the caller's data directly, and to modify that data if the called function chooses to do so.

int &i = i is reference to int

* pointers enables an alternate form of pass-by-reference in which the style of the call clearly indicates pass-by-reference (and the potential for modifying the caller's arguments).

squareReference (const int& i)

↳ specify a reference to a constant (nonmodifiable)

* reference variables must be initalized in their declarations and cannot be reassigned as aliases to other variables.
* a reference argument must be an lvalue (e.g., a variable name)

Default Arguments: default values are taken by default whenever a function call is made without default arguments.

- must be rightmost (trailing) arguments in parameter list.
- default values can be constants, global variables, function calls
- can also be used with inline functions
- should be specified with the first occurrence of the function name—typically, in the function prototype.

```
int box( int length, int width, int height = 1 );
```

- Unary scope resolution operator (::) enables a program to access a global variable when a local variable of the same name is in scope.

```
:: i
```

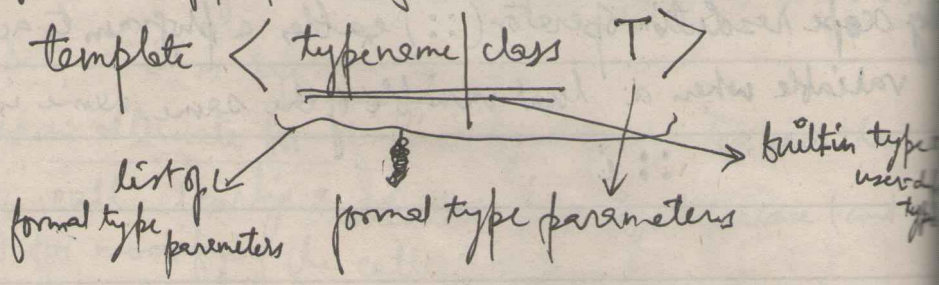
Function Overloading: same name functions with different number, types, order of arguments

* [signature = function name + parameter types (in order)]

- overloaded functions can have different return types, but must have different parameter lists.

Function Templates: for automatically generating overloaded functions that perform identical tasks on different data types.

- a means of code generation
- (used to perform similar operations on different data types)
- defines a whole family of solutions (automatically generates seperate function-template specializations to handle each type of call appropriately)



* define size of an array as a constant variable

* ~~define size of constant~~

---

char arrayname[number of characters + terminating null character];

char a[20] ≡ char a[19+1]

* string = an array of characters

char string1[] = "first";

char string1[] = {'f', 'i', 'r', 's', 't', '\0'};

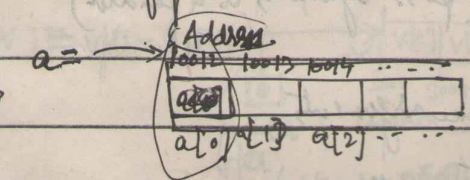
* a space terminates input string.

* char s[80];
  cin >> s ← "Hello World"
  cout << s → Hello

* static local array initializes element to 0 first time the function is called.

* to pass an array argument to a function, specify the name of array without any bracket. (and optionally, we should pass the size of array also in the argument)

* C++ passes arrays to functions using simulated pass-by-reference (called function can modify the element values in the caller's original arrays.)

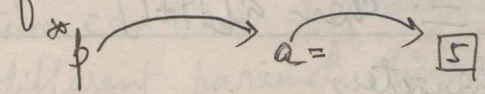


* individual array elements are passed by value exactly as simple variables are (such simple piecy data are called scalar or scalar quantities)

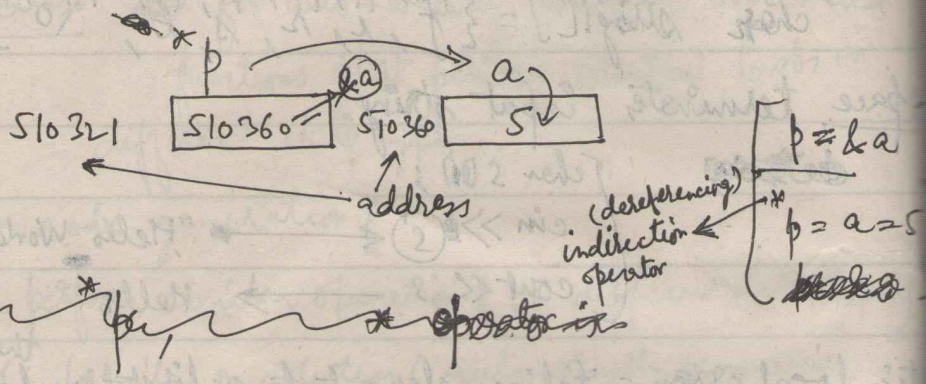
void modifyElement(int e)
{ .... }

⊛ const can be used to prevent modification of array values in a function.

⊛ pointer indirectly references a value. (indirection)



⊛ pointer must be initialized to 0, NULL, or an address.



⊛ the & and * operators are inverses of one another.

$$\&{}^{*}x = {}^{*}\&x$$

$$^{*}p = a$$
$$p = \&a$$

⊛ Calling functions by reference (pass arguments to a function)

1. pass by value
2. " " reference with reference arguments
3. " " " " pointer "

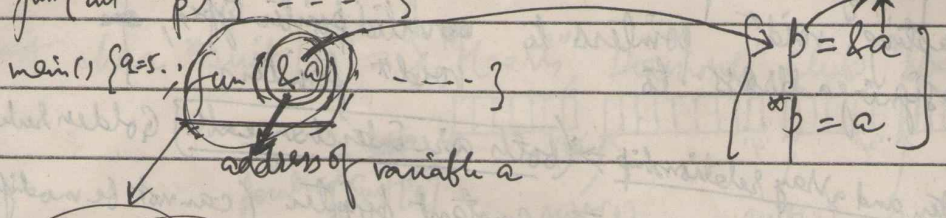
⊛ pointers, like references, also can be used to modify one or more variables in the caller or to pass pointers to large data objects to avoid the overhead of passing the objects by value.

④ the name of array is the starting location in memory of the array
(the array name is already a pointer)

arrayname ≡ &arrayname[0]

→ int fun(int *p) { --- }

int main() { a=5; fun(&a); --- }



fun(&a) ≡ fun(p)

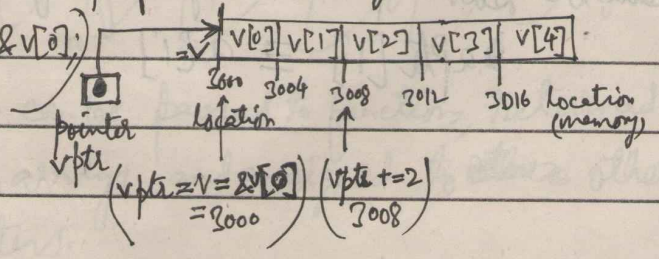
⇒ ⑤ array ⇒ stores related data items of the same type
structure ⇒ " " " " " " different types

⑥ operator sizeof( ) can be applied to any variable name/type name/constant value.
it returns the number of bytes used to store the " " "

Pointer expressions and pointer arithmetic (++, --, +, +=, -, -=)

⑦ pointer arithmetic is performed on array's elements.

⑧ (vptr = V; ≡ vptr = &V[0];)



⑨ Assign a pointer of one type to a pointer of another (other than void*) with casting the first pointer to the type of the second pointer.

⑩ without casting ⇒ (int*, double*, float*, ...) (void*)

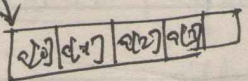
not allowed

* A void* pointer can not be dereferenced! (unknown type of memory)
* pointers can be compared using equality and relational operators. (of same array)
* All operations on a void* pointer are syntax errors, except

valid
- comparing void* pointers with other pointers,
- casting void* pointers to valid pointer types,
- assigning address to void* pointers

* pointer and array relationship ⇒ (both are interchangeable) Golden rule *p = a
  * array name ≡ constant pointer (can not be modified)

✓ aPtr = a ≡ &a[0]

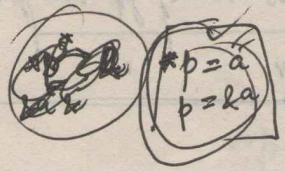


[aPtr+3 ≡ &a[3] ≡ a+3 ≡ &a[3]]

pointer/offset notation for element position

*(aPtr+3) ≡ a[3] ≡ *(a+3)

aPtr+3 ≡ &a[3]



* array name can be treated as pointer and used in pointer arithmetic

*(a+3) ≡ a[3]

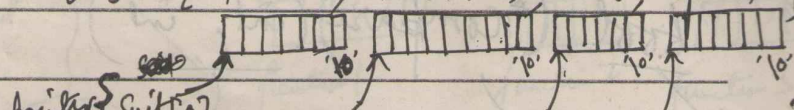
* pointers can be subscripted exactly as arrays can.

aPtr[1] ≡ a[1] (pointer/subscript notation)

Array of pointers (Arrays containing pointers)

e.g. array of strings: (string array) where each entry in an array of strings is actually a pointer to the first character of a string.

```
const char *suit[4] = {"Heart", "Diamonds", "Clubs", "Spade"};
```



pointer { suit[0], suit[1], suit[2], suit[3] }

suit array pointer point to const char

Function Pointers: (a pointer to a function contains the address of the function in the memory)

e.g., similarly { array name a ≡ the address of first element of array in memory
function name f ≡ " " " " starting of the code "

(that performs the function's task)

a ≡ &a[0]

f ≡ &f[0]

passing pointer to function as an argument in function

returning " " " " "

Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other function pointers.

In C#, Delegate ≡ Function pointer

e.g.

```
void shuffle(int[], const int, bool (*)(int, int));
```

essential

```
bool (*compare)(int, int)
```

return type of

pointer to function

parameter list of the function

```
return type (*pf) (argument, ...)
```

parameter list

essential parentheses

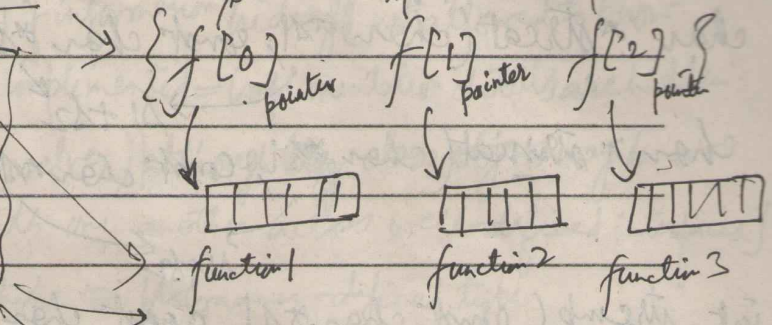
pointer to function ≡ alias/nickname of function

e.g.

```
bool ascending(int a, int b) ≡ bool (*Compare)(int a, int
```

Array of Pointers to Functions:

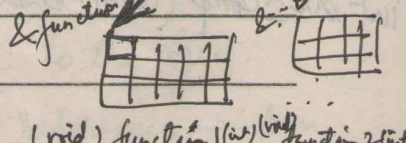
void (*f[3])(int) = {function1, function2, function3};



① all the functions to which the array points must be have the same return type and the same parameter type

(elements of array must be have same return types)

returntype (*pfunction[array])(parameters,...) = {pfunction[0], pfunction[1]}



functions (void) function1(int) (void) function2(int)

② By indicating the subscript into the array of function pointers, and the pointer in the array is and with same return types parameter types

Characters & Strings =

① character constants is an integer value represented as a character in a single quote.

② string literals = string constants are written in double quote,

③ a string is an array of characters ending in the null character ('\0').

④ a string is a constant pointer to the string's first character like array

strings = arrays of.

⑤ string may be assigned in a declaration by

① char a[] = "blue";

② const char *pa[] = "blue";

#include <cstring>

char *strcpy(char *s1, const char *s2);

char *strncpy(char *s1, const char *s2, size_t n);
→ at most n characters

char *strcat(char *s1, const char *s2);
→ s1+s2

char *strncat(char *s1, const char *s2, size_t n);
→ at most n characters
→ s1+s2

int strcmp(const char *s1, const char *s2);
→ (compare)
(s1 == s2) → =0
(s1 < s2) → <0
(s1 > s2) → >0

int strncmp(const char *s1, const char *s2, size_t n);
→ upto n characters
→ (compare) upto n characters of both

size_t strlen(const char *s);
→ length of string s (number of characters)
size_t → unsigned int / long