Identifiers are used to name **macros**, variables, **functions**, **structs**, **unions**, classes, member variables, member functions, and other entities in a computer program

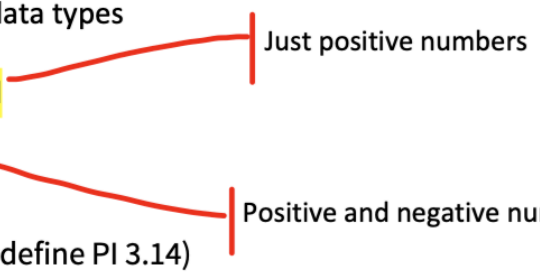
Reserved keywords are keywords that the already have a reason to be in the program such as int, double, short, const, continue etc

Reserved keywords **cannot** be used as identifiers!

C++ only data types are bool (boolean) and wchart_t (wide character)

Use sizeof() to find out size of different data types

c/c++ integers can be signed or unsigned




C/C++ has 2 types of constants:
- Literal (eg int i = 0)
- Symbolic (eg const float PI = 3.14; or #define PI 3.14)

```
i = (int)d; // explicit type casting
```

In c++ static_cast performs explicit type conversion

```
EG.
static_cast<int>(7.5) // evaluates to 7
static_cast<float>(5/2) // evaluates to 2.0
```

Another way is to use type(expression)

```
EG.
int(7.5) // evaluates to 7
```

- C/C++ specific operators
  - Pointers and reference related operators (*, &, ->)
  - Others (sizeof, scope, casting)

Operator *precedence* determines the sequence in which operators in an expression are evaluated

- *Associativity* determines execution for operators of equal precedence
- Precedence can be overridden by explicit grouping using ( and )

In C/C++, the condition is an expression that evaluates to any type
• Considered true if expression evaluates to non-zero value, otherwise false

EG

```
int i = 100;
while (i--) { // do stuff
}
```

Unlike Java, C/C++ allows functions to exist on their own, i.e., outside any class
• In C, functions are first-class entities: a C program consists of one or more functions

• A C/C++ program must have exactly one main function

Before a function can be invoked, either the **function definition** or **function prototype** should have been declared prior to the invocation
• **Function prototype** – declaration specifying the return type, function name, and list of parameter types

```
  return_type function_name
 ( parameter_types_list );
```

Each function prototype needs to have a function definition - sets up the behaviour of the function

- A typical C/C++ program consists of
  - 1 or more **header** files
  - 1 or more C/C++ **source** files

```
#include <stdio.h>

int main(void)
{
   printf("Hello world\n");

   return 0;
}
```

*Preprocessor* directive to include `stdio.h` header file which contains `printf` function *prototype*

`main` function *definition*, invoking `printf` to display "Hello, world", and return 0

Hello world using pure C

`#include "filename"`
Include file named `filename`

- Preprocessor searches for file in current directory first, then in locations specified by programmer

A header file usually contains function prototypes, constant definitions, type definitions, etc.

In C++ input and output is performed in the form of sequence of bytes or more commonly known as **streams**.

**Header files available in C++ for Input – Output operation are:**

- **iostream**: iostream stands for standard input output stream. This header file contains definitions to objects like cin, **cout**, cerr etc.
- **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

For using cin and cout we must include the header file *iostream* in our program.

The insertion operator is represented by **<<**

**EG.**

```
#include <iostream>

using namespace std;

int main( ) {
   char sample[] = "GeeksforGeeks";

   cout << sample << " - A computer science portal for geeks";

   return 0;
}
```

Will output: GeeksforGeeks - A computer science portal for geeks

Using the namespace std at the beginning means we don't have to write 'std' each time we use the cout object, other the insertion line would be:

*std::cout << sample << " - A computer science portal for geeks";*

**Macros**: Macros are piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The '#define' directive is used to define a macro.

EG.

```
#include <iostream>

// macro definition
#define LIMIT 5
int main()
{
    for (int i = 0; i < LIMIT; i++) {
        std::cout << i << "\n";
    }

    return 0;
}
```

Macros can be used to define function-like macros

EG.

```
#define READ_CHAR() getchar()

int c = READ_CHAR();
```

- Just like functions, function-like macros can take arguments
  - Insert comma-separated parameter names between ( and )
  - Parameter names must be valid identifiers

```
#define max(X, Y) ((X) > (Y) ? (X) : (Y))
```

- Invoke just like normal functions

```
z = max(1, 3);
```

```
z = ((1)>(3)?(1):(3));
```

This expression evaluates to **3**

Be careful when using these and consider BEDMAS. A way to get around this is by enclosing individual variables with (), including the whole replacement text

EG.

```
#define SQ(X) ((X) * (X))
```

Identifier scope refers to parts of the program where an identifier is accessible or visible

- **Local**: identifiers declared within a function or block - only visible inside the block
- **Global**: identifiers declared outside functions - visible from the line of declaration to the end of file

```
extern int a; defines a global variable a
```

## Difficulties with Global Identifiers

- When a header file is included in a program, all global identifiers in the header file also become global identifiers in the program
- When program has global identifiers with same name as in header file, compiler will generate an error (e.g., "identifier redefined")

- Can be solved using the **namespace** mechanism

Members can be constants, variables, functions, classes, or another namespace

Two ways to access a namespace member outside its namespace:
- Use `namespace_name::identifier` syntax
- Use the `using` keyword to access specific or all members of a namespace

`The scope resolution operator is ::`

## Arrays

- An array is a collection of data that holds a fixed number of data (values) of the **same type**
- In C/C++, arrays and pointers are closely related concepts
  - An array name by itself is treated as a *constant pointer*
- We distinguish between two types of arrays:
  - One-dimensional arrays
  - Multi-dimensional arrays

- The C/C++ language places no limits on the number of dimensions in an array, though specific implementations may

Example:

– We declare an array named `data` of `float` type and size 4 as:

```
float data[4];
```

– It can hold 4 floating-point values

• The size and type of arrays **cannot** be changed after their declaration!

# Initializing Arrays

- Arrays can be also initialized when they are **declared** (just as any other variables):

```
float data[4] = {22.5, 23.1, 23.7, 24.8};
```

- An array may be **partially initialized**, by providing fewer data items than the size of the array

```
float data[4] = {22.5, 23.1};
```

– The remaining array elements will be automatically initialized to zero

- If an array is to be completely initialized, the dimension (size) of the array is not required

```
float data[] = {22.5, 23.1, 23.7, 24.8};
```

– The compiler will automatically size the array to fit the initialized data

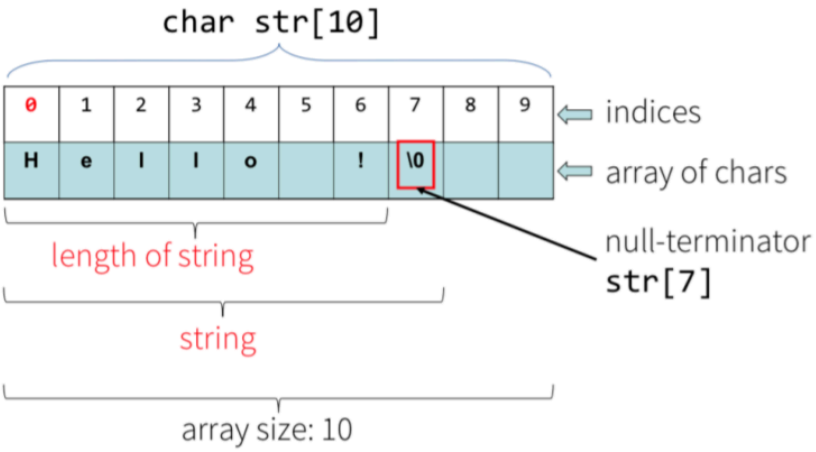
sizeOf() will give the size of an array

- It will return the *number of bytes the array "occupies" in the memory*
- To determine the number of elements in the array, the returned value must be divided by the number of bytes reserved for the data type !

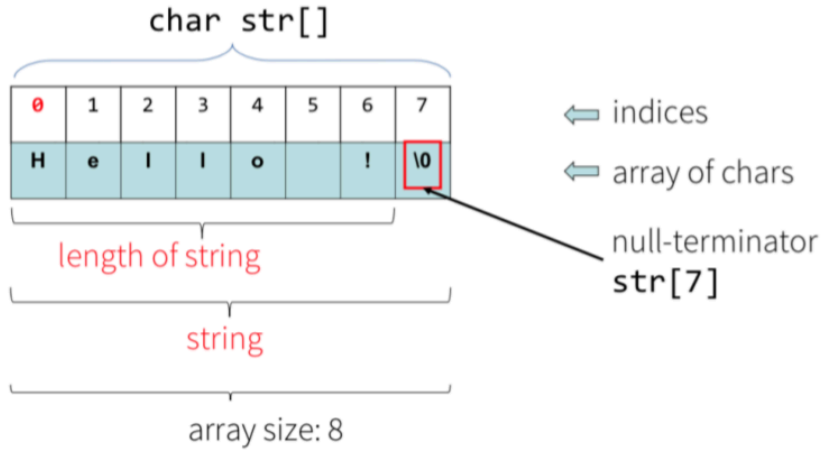
A character array that contains ASCII characters terminated by the null character **'\0'** is a C string variable

Helpful examples to understand what's going on:

```
char str[10] = "Hello !";
```




```
char str[] = "Hello !";
```



# 2D Arrays

- Declaring a char array with 3 rows and 5 columns

```
char two_d[3][5];
```

  - The array can hold 15 char elements
- Accessing a value

```
char ch;
ch = two_d[2][4];
```

- Modifying a value

```
two_d[0][0] = 'x';
```

- The array can be initialized in one of the following ways

```
int two_d[2][3] = {{5, 2, 1}, {6, 7, 8}};
int two_d[2][3] = {5, 2, 1 , 6, 7, 8};
int two_d[][3] = {{5, 2, 1}, {6, 7, 8}};
```

  - The number of columns must be explicitly stated. The compiler will find the appropriate amount of rows based on the initializer list

Declaring a three-dimensional (3D) array

```
float three_d[2][4][3];
```

– Here, three_d can hold 24 elements. Each 2 elements have 4 elements, which makes 8 elements and each 8 elements can have 3 elements.

**Enumeration** (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

```
enum enum_tag {
      identifier_0, identifier_1, ..., identifier_n
} variable_list;
```

- *enum_tag* specifies the name of the enumeration type
- *enum_tag* and *variable_list* are optional
- If *enum_tag* is not specified, *variable_list* should be specified; otherwise, there is no way to declare variables using the unnamed enumeration type

The identifiers in the braces are symbolic constants that take on integer values from `0` through n
➔ `identifier_0hasvalue0,`
➔ `identifier_1` has value 1, and so on

EG.
```
enum colors { red, yellow, green }; // where red = 0, yellow = 1, green = 3
```

It is possible to override the integer assignment, e.g.
```
enum colors { red = 3, yellow = 2, green = 1};
```

```
If enum colors { red, yellow = 3, green, blue }; then the program will
continue the progression from the assigned value so red = 0, green = 4 and
blue = 5
```

A **structure** is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type (also known as a tuple).

- struct declaration that only defines a type:
```
struct student_info { // named struct
      char name [20];
      int student_id;
      int age;
}; // does not reserve any space
```
- struct declaration that defines a type and reserves storage for variables:
```
struct student_info { // named struct
      char name [20];
      int student_id;
      int age;
} s, t; // reserves space for s and t
```

**typedef** : typedef is used to give data type a new name, for example:

```c
// C program to demonstrate typedef
#include <stdio.h>

// After this line BYTE can be used
// in place of unsigned char
typedef unsigned char BYTE;

int main()
{
    BYTE b1, b2;
    b1 = 'c';
    printf("%c ", b1);
    return 0;
}
```

We can access components of a struct by using a dot (.) also known as a direct component selection operator which has level 1 priority in the operator precedence

• When a structure variable is passed as an input argument to a function, all its component values are copied into the local structure variable

- An array of structures can be defined as follows:

```c
typedef struct {
   int student_id;
   double gpa;
} student_t;

student_t student_list[50];

student_list[3].student_id = 300922023;
student_list[3].gpa = 8.0;
```

Like structures, a **union** is a user defined data type. In union, all members share the same memory location.

EG.

```
union test {
    int x, y;
};
```

x and y in test both share the same location in memory which means if we say:

```
union test t;

    t.x = 2; // then t.y also gets value 2
```

The size of a union is determined by the largest member

```
sizeof(union space) =
max(sizeof(member1),sizeof(member2),...)
```

To find out what a union contains (because it doesn't know), Use another variable to hold that info (eg an if check if a union component is a char or an int)

Using **tagged unions** is a way to pair the type info together with the union – implicit in other programming languages like Java. We use a global enum to do this which essentially assigns each value to a case in the enum (eg int or char).

The difference of C++ structures is that C++ structures can have functions as members and can be extended (supports inheritance) **but a key disadvantage is that all the functions and member variables are public**

C++ classes generalizes structures in an object-oriented sense

A class is a collection of fixed number of components called **members** of the class

```
class Time {
public:
      void set(int, int, int);
      void print() const;
      Time();
      Time(int, int, int);

private:
      int hour;
      int minute;
      int second;
};
```

**Member access specifiers**

Possible specifiers:
- `private`
- `protected`
- `public`

**Private**: The class members declared as **private** can be accessed only by the functions inside the class.
**Protected**: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.
**Public**: All the class members declared under public will be available to everyone.

When class performs dynamic memory allocation, **destructor** is also needed

**`const`** at the end of function specifies that member function cannot modify member variables

When member access specifier is not indicated, default access is **`private`**

- Member functions can be declared in 2 ways:
  - By specifying the function prototype
  - By specifying the function implementation
- How about in Java?

```
class Time {
public:
      void print() const;
      void set(int h, int m, int s) {
            hour = h;
            minute = m;
            second = s;
      }
      Time();
      Time(int, int, int);

private:
      int hour;
      int minute;
      int second;
};
```

**Inline** function is one of the important feature of C++. The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called. Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime.

- For member functions that are not implemented in the class declaration, they must be implemented separately

```
class Time {
public:
  void print() const;
  void set(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
  }
  …
};
```

```
#include <cstdio>

void Time::print() const
{
  printf("%2d:%2d:%2d", hour,
    minute, second);
}
```

## Another Syntax For Initializing Member Variables

```
Time::Time() {
    hour = 0;
    minute = 0;
    second = 0;
}

Time::Time(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
}
```

```
Time::Time() : hour(0), minute(0),
    second(0) {
}

Time::Time(int h, int m, int s) :
    hour(h), minute(m),
    second(s) {
}
```

- You may declare classes and implement the member functions in the same C++ source file
- Disadvantage: other sources will not be able to use the class

- Good programming practice is to declare the class in a header file
- Separate the implementation of the member functions (and possibly constructors) in another source file

A static member variable is a variable that is shared by all instances of a class

- Often used to declare class constants
- A static member function is function that can be invoked outside class instance
- Member functions and variables can be made static by using the `static` qualifier

C++ supports class **inheritance**

**Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
**Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.
- **The sub class is said to extend the base class**

**Example of a class extending another**

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

- *subclass_name* is the identifier given to the sub class being declared
- *access_mode* controls the access of inherited fields
- *baseclass_name* is the identifier of the super class being extended

```
class Dog : public Animal {
public:
  int bark(int loudness);
  int bite(int strength);
  void run(int speed);
  void eat(int food);
  Dog(const char *n) : Animal(n) {}

private:
  int skills;
};
```

Constructor of Dog invokes appropriate super class constructor

- Unlike Java, C++ does not have super keyword for invoking super class constructor

Declare the member function to be overridden in the class declaration
- Prototype must be exactly the same as member function to override

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called **abstract class**.

For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes. Abstract classes cannot be instantiated.

A pure virtual function (or abstract function) in C++ is a **virtual function** for which we don't have implementation, we only declare it. Virtual functions must be implemented.

**Multiple Inheritance** is a feature of C++ where a class can inherit from more than one classes.

EG.

- Suppose that Human and Dog are existing classes

```
class Werewolf : public Human, public Dog {
public:
  void transform();
  …
  Werewolf(const char *n) : Human(n), Dog(n) {}

private:
  int transformCount;
  …
};
```

- What happens if base classes of a derived class have a common member function?

```
class A : {
public:
  void foo();
  …
};

class B : {
public:
  void foo();
  …
};
```

```
class C : public A, public B {
  …
};
```

Class C must override foo(), example:

```
class C : public A, public B {
public:
  void foo() {
    A::foo(); // Use A's implementation of foo!
  }
};
```

## Recap Strings in C

- C does not support strings as a basic data type
- A string is a sequence of characters that is treated as a single data item and terminated by a null character also known as the null-terminator, null byte or just '\0'
- In C a string is actually a one-dimensional array of characters
- There are functions defined to manipulate strings in string.h
- The cstring library can be used in C++
- Code safety / security is the responsibility of the programmer

**strcpy(s1, s2);**
Copies string s2 into string s1.

**strcat(s1, s2);**
Concatenates string s2 onto the end of string s1.

**strlen(s1);**
Returns the length of string s1.

**strcmp(s1, s2);**
Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

**strchr(s1, ch);**
Returns a pointer to the first occurrence of character ch in string s1.

**strstr(s1, s2);**
Returns a pointer to the first occurrence of string s2 in string s1.

- C++ has a string class type that implements a programmer defined string datatype
- Similar to Java String class
- There are multiple constructors to instantiate a string object
- a wide range of operators and member functions are available for variables declared as string type

Memory allocated to a program includes space for machine language code and data

- Text / Code Segment
  - Contains program's machine code
- Data spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Heap Segment** – For dynamically allocatedmemory; expands / shrinks as program runs
  - **Stack Segment** – For temporary data, *e.g.*, local variables in a function; expands / shrinks as program runs and functions are called


- All information accessible to a running computer program must be stored somewhere in the computer's memory.
- C provides the ability to access specific memory locations, using "**pointers**".
- Memory locations are identified by their ***address***.

To use pointers in C, we must understand below two operators.

- To access address of a variable to a pointer, we use the unary operator **&** (ampersand) that returns the address of that variable. For example &x gives us address of variable x.

One more operator is **unary *** (Asterisk) which is used for two things :

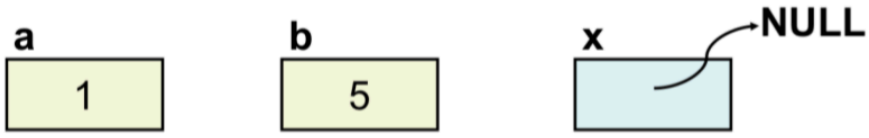
- To declare a pointer variable: When a pointer variable is declared in C/C++, there must a * before its name.

To access the value stored in the address we use the unary operator (*) that returns the value of the variable located at the address specified by its operand.

**x = &a;**

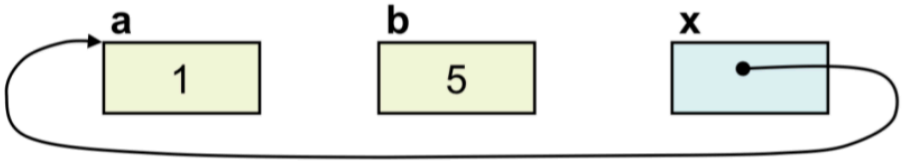
- **x** will now point to **a**, i.e., **x** stores **a**'s address

Declaration: **int a = 1, b = 5; int *x = NULL;**




**NULL** – pointer literal/constant to non-existent addr.

Assignment: **x = &a;**




• Arrays in C are pointed to, i.e. the variable that you declare for the array is actually a pointer to the first array element
• You can interact with the array elements either through pointers or byusing[]

**int z[], *ip;**
**ip = &z[0];**
**z[0]**, ***ip** or ***z** can all be used to access the first element of the array **z[]**

```
int (*ptr)[10];
```

Here *ptr* is pointer that can point to an array of 10 integers.

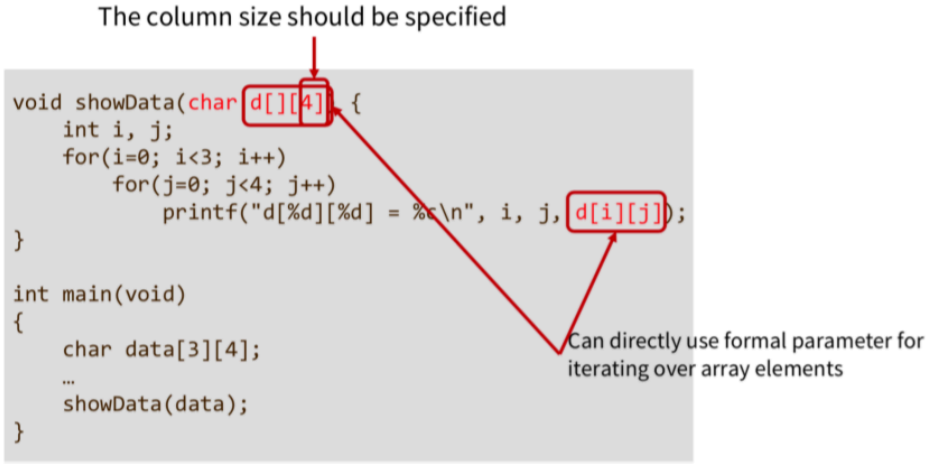
When passing an array as an argument to a function, it is passed by its **memory address** (starting address of the memory area) and not its value!

• 1D array can also be passed using pointer notation

```
float average(int *a) {
      int sum = 0;
      for (int i = 0; i < 6; ++i)
            sum += a[i];
```

Can directly use formal parameter for iterating over array elements

This applies to multi-dimension arrays too.




We already know that a pointer points to a location in memory and thus used to store address of variables. So, when we define a pointer to pointer (or a double pointer), the first pointer is used to store the address of second pointer. That is why they are also known as double pointers.

- 2D array can also be passed using double pointer notation
- Will require auxiliary array of pointers to refer to array elements

Indirection /dereferencing operator *

***myptr = y;** //dereference myptr and assign the value from y

## Pointer Arithmetic

- Addition and subtraction can be performed on pointers , this is useful to iterate through arrays
- ++ Increments the pointer to the next element in the array
- -- decrements the pointer to the previous element in the array
- + adds the specified number of positions in the array to the pointer eg. pointer + 4 moves 4 elements in the array
- - subtracts the specified number of positions in the array from the pointer
- You can also add or subtract two pointers of the same type

Functions can also return pointers. But don't return pointers to local variables.

**Const pointer:** cannot be reassigned to point to a different location from the one it is initially assigned, but it can be used to modify the location that it points to

```
Datatype * const p;
```

**Pointer to a const location:** can be reassigned to point to a different location, but it cannot be used to modify any location that it points to

```
const Datatype * p;
```

**Const pointer to a const location:** can neither be reassigned nor used to modify the location that it points to

```
const Datatype * const p;
```

It is inappropriate to assign an address of one type of variable to a different type of pointer.

Example:

```
int V = 101;
float * P = &V; // Generally results in a Warning not an error
```

When assigning a memory address of a variable of one type to a pointer that points to another type, it is best to use the cast operator to indicate the cast is intentional (this will remove the warning), but it is still unsafe to do this !!!

Example:

```
int V = 101;
float * P = (float *) &V;   // Casts int address to float
```

A **void *** is considered to be a general pointer
No cast is needed to assign an address to a **void *** or from a **void *** to another pointer type

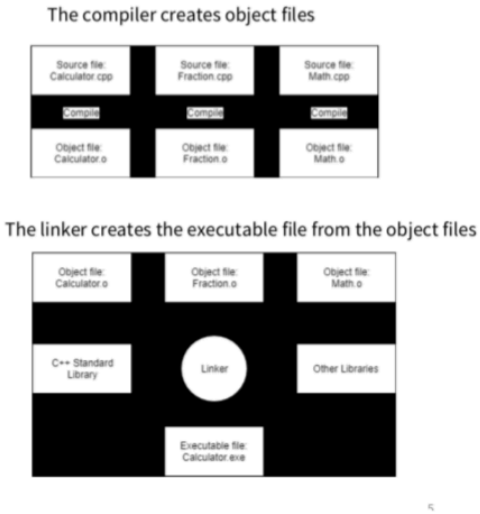
new and delete reserved words are used to create and destroy dynamic variables
• The new keyword allocates memory for the variable and returns a pointer to it. The following syntax allocates memory for a single variable and an array of variables.
new *datatype*;
new *datatype* [an expression that evaluates to an integer];

Using new and delete in a pair prevents memory leaks (memory leaks occur when programmers create a memory in heap and forget to delete it.)

The delete keyword is used to return / release the memory that was allocated using the new keyword. The following syntax deallocates memory for a single dynamic variable and a dynamic array variable
delete pointervariable;
delete [] pointervariable;

Recap:

- C Header files used by the Include statement
  - Include is a preprocessor directive
  - Instructs the compiler to read the source code from another file, this source code is then included in the code being compiled
  - The include statement is not limited to library header files such as <stdio.h> it can also include any file you specify




C storage classes are:
• **Auto (is the default) • static**
• **register**
• **extern**
Storage class of a variable determines its:
• ***Scope*** attribute – where is a variable visible
• ***Lifetime*** attribute – how long does a variable exists

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

- Lifetime/storage attributes can be:
  - `static` variables are allocated memory when program starts;
  - `auto` – automatic variables are allocated memory when execution enters the block that contains it;
  - `register` – reside in CPU's high speed memory
- Scope attributes can be:
  - `local` – `v` is only visible inside the current, innermost scope, independent of storage/lifetime attribute; e.g. there are `local static` variables in C
  - `global` – `v` is visible in the whole compilation unit, from the line of declaration to the end of file
  - `external` – `v` is visible in all compilation units; `static`

• **auto** is the default storage class for a variable defined inside a function body or a statement block
• **auto** prefix is optional; i.e. any locally declared variable is automatically **auto**, unless specifically defined to be static

- *Automatic variables* may *only* be declared *within* functions and compound statements {*blocks*}
  - Storage *allocated* when function or block is entered
  - Storage is *released* when function returns or block exits
- Parameters and result are similar to automatic variables
  - Storage is *allocated* and *initialized* by *caller* of function
  - Storage is *released* after function *returns* to caller.
- Variables declared within a function or compound statement are visible *only* from the point of declaration to the end of that function or compound statement.

**Geeks for Geeks explanation of these:**

**auto:** This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

**extern** : Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

**static:** This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

**register**: This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

*More info on the slides about each of these.*