

C Programming

History

- **Dennis Ritchie** designed the C language at **Bell Laboratories** in the early 1970s
- Ancestral languages:
 - ALGOL 60 (1960)
 - Cambridge's CPL (1963)
 - Martin Richards's BCPL (1967)
 - Ken Thompson's B language (1970) at Bell Labs
- C is a general-purpose programming language
- UNIX operating system was originally written in C

Overview

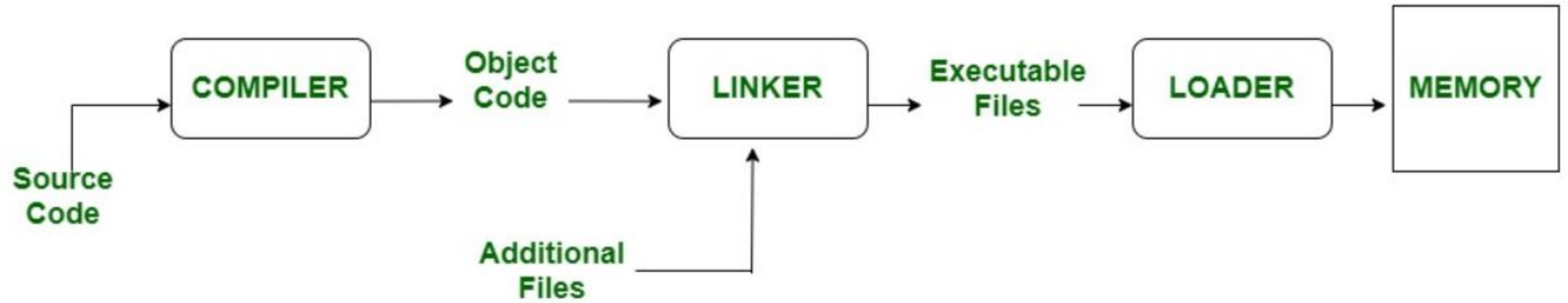
- C program is composed of
 - One or more **source files**, or
 - Translation units
 - Common declarations are often collected into **header files** and are included into the source files with a special `#include` command
 - One external function must be named **main()**; this function is where the program starts
- Each of which contains some part of the entire C program - typically, some number of external functions

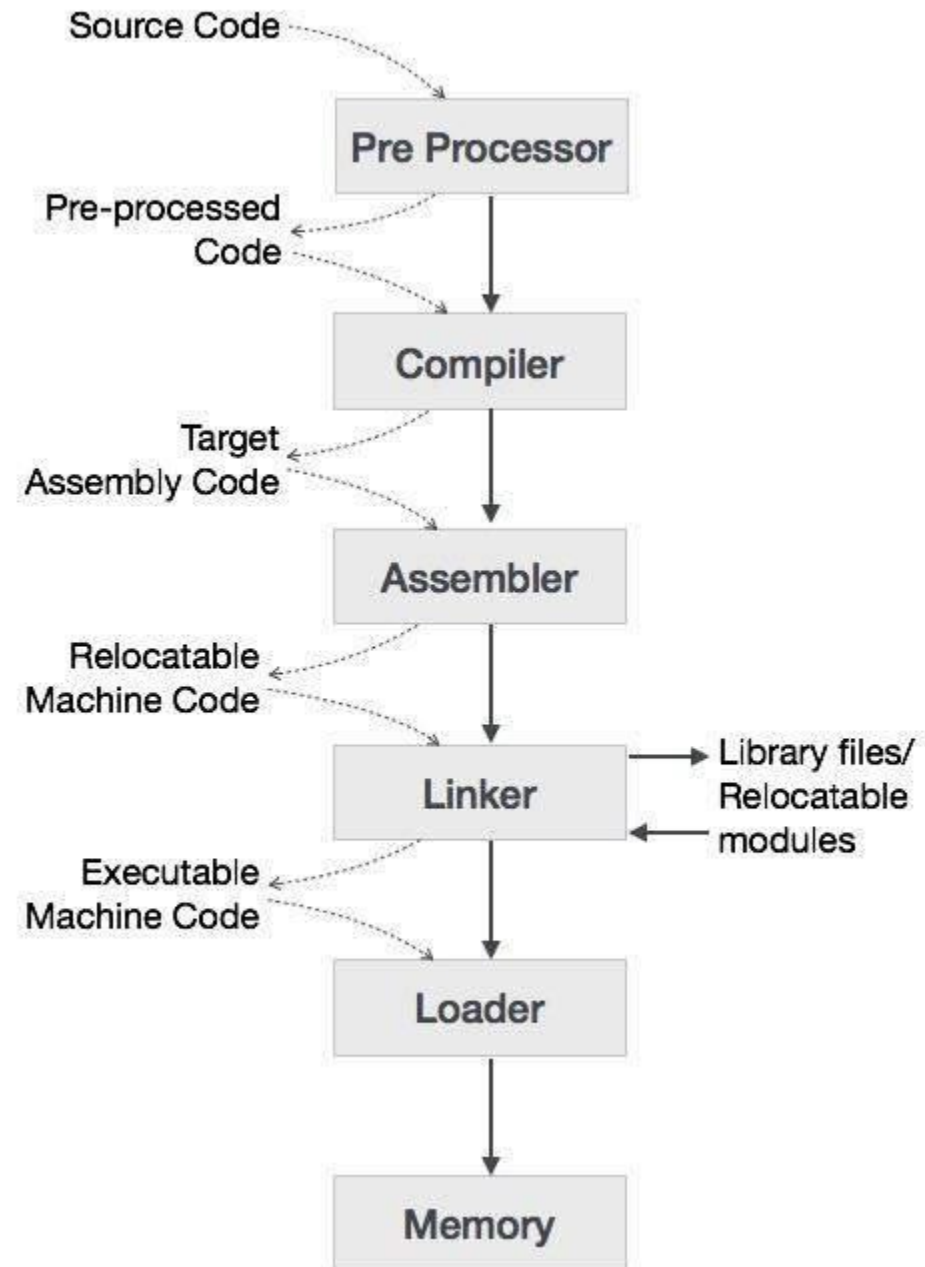
C Compiler

- Independently processes each source file and translates the C program text into instructions understood by the computer
- “Understands” the C program and analyzes it for correctness
 - If the programmer has made an error the compiler can detect, then the compiler issues an error message
- The output of the compiler is usually called object code or an object module

Linker

- When all source files are compiled, the object modules are given to a program called the linker
- It **resolves references** between the modules, **adds functions** from the **standard run-time library**, and detects some programming errors such as the failure to define a needed function
- This is typically not specific to C; each computer system has a standard linker that is used for programs written in many different languages
- It produces a single executable program, which can then be invoked or run





Lexical Elements – Character Set

A B C D E F G H I J K L M N O P Q R S T
U V W X Y Z a b c d e f g h i j k l m
n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

the SPACE

Char	Name	Char	Name	Char	Name
!	EXCLAMATION MARK	+	PLUS SIGN	"	QUOTATION MARK
#	NUMBER SIGN	=	EQUALS SIGN	{	LEFT CURLY BRACKET
%	PERCENT SIGN	~	TILDE	}	RIGHT CURLY BRACKET
^	CIRCUMFLEX ACCENT	[LEFT SQUARE BRACKET	,	COMMA
&	AMPERSAND]	RIGHT SQUARE BRACKET	.	FULL STOP
*	ASTERISK	'	APOSTROPHE	<	LESS-THAN SIGN
(LEFT PARENTHESIS		VERTICAL LINE	>	GREATER-THAN SIGN
_	LOWLINE (underscore)	\	REVERSE SOLIDUS (backslash)	/	SOLIDUS (slash, divide sign)
)	RIGHT PARENTHESIS	;	SEMICOLON	?	QUESTION MARK
-	HYPHEN-MINUS	:	COLON		

the horizontal tab (HT),
vertical tab (VT), and
form feed (FF) control characters

Tokens

- The characters making up a C program are collected into lexical tokens
- Classes of tokens: **operators**, **separators**, identifiers, **keywords**, and **constants**
- The compiler always forms the longest tokens possible as it collects characters in left-to-right order, even if the result does not make a valid C program
- Adjacent tokens may be separated by whitespace characters or comments

Characters	C Tokens
forwhile	forwhile
b>x	b, >, x

Keywords

- Keywords are predefined, reserved words used in programming that have special meanings to the compiler
- Keywords are part of the syntax, and they cannot be used as an identifier

Keywords

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int result;
```

```
    if ( result != 0 )
```

```
        printf_s( "Bad file handle\n" );
```

```
}
```

Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
continue	for	signed	void
do	if	static	while
default	goto	sizeof	Volatile
const	float	short	Unsigned

Identifiers

- "Identifiers" or "symbols" are the names supplied for variables, types, functions, and labels in a program by the user
- Identifier names must differ in spelling and case from any keywords
- Identifier can be created by specifying it in the declaration of a variable, type, or function
- Once declared, you can use the identifier in later program statements to refer to the associated value

Identifiers

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int result;
```

```
    if ( result != 0 )
```

```
        printf_s( "Bad file handle\n" );
```

```
}
```

Constants

- C Constants are like a variable, except their value never changes during execution once defined
- Types
 - Integer
 - Decimal
 - Hexadecimal – 0x, 0X
 - Octal – 0
 - Float
 - Character
 - String

Pre-processor

- The C preprocessor is a simple macro processor that conceptually processes the source text of a C program before the compiler proper reads the source program

#define

Define a preprocessor macro.

```
#define sum(x,y)  x+y  
result = sum(5,a*b);
```

```
#define BLOCK_SIZE 0x100
```

#include

Insert text from another source file.


#pragma

Specify implementation-dependent information to the compiler.

C Program - Introduction


```
#include <stdio.h>
```

```
void main(void) {  
    printf("Hello world!\n");  
}
```



hello.c

source code



C compiler

hello.exe

main()

```
#include <stdio.h>
```

```
void main(void) {  
    printf("Hello world!\n");  
}
```

starting point of the program

Function Prototyping

```
int main(int argc, char *argv[])
```

Return type

Function Name

Input Arguments

Getting Inputs from console

Header File

```
#include <stdio.h>
```

```
void main(void) {  
    printf("Hello world!\n");  
}
```



stdio → Standard Input Output

Contains functions that will be used in the source code

#include → Pre-processor directive
Instructs compiler to find the function definition of certain functions mentioned in the header file

The preprocessor is a program that runs before the C compiler itself
It serves to perform text substitutions on the source code prior to the actual compilation

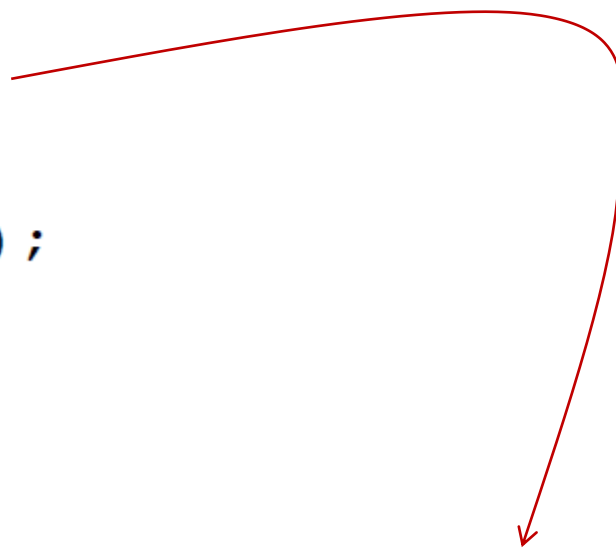
Standard C Library files

C Header Files	Description
<assert.h>	Program assertion functions
<ctype.h>	Character type functions
<locale.h>	Localization functions
<math.h>	Mathematics functions
<setjmp.h>	Jump functions
<signal.h>	Signal handling functions
<stdarg.h>	Variable arguments handling functions
<stdio.h>	Standard Input/Output functions
<stdlib.h>	Standard Utility functions
<string.h>	String handling functions
<time.h>	Date time functions

Short form

```
#include <stdio.h>
```

```
void main(void) {  
    printf("Hello world!\n");  
}
```



```
#include <stdio.h>  
void main(void) {printf("Hello world!\n");}
```

Declaration

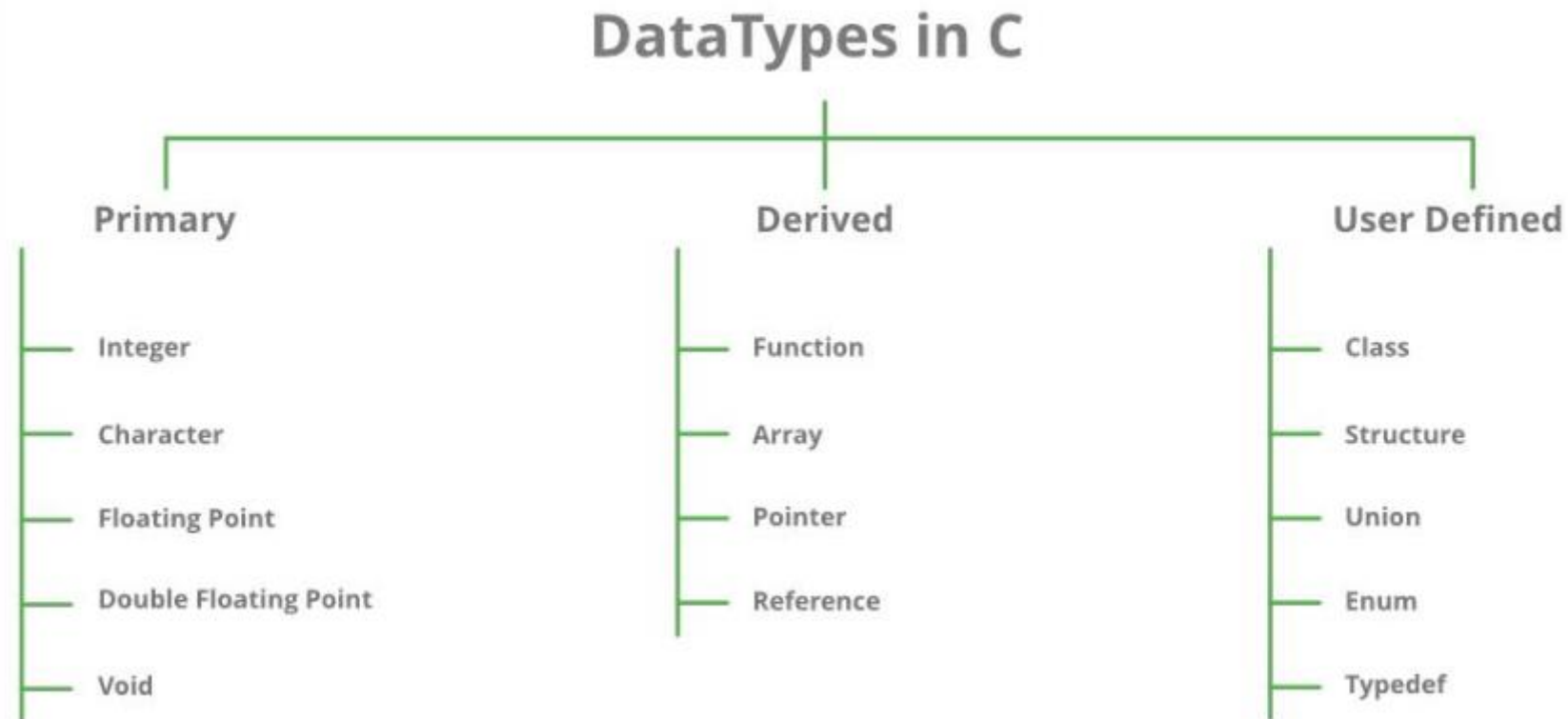
- A declaration is a statement *describing an identifier*, such as the name of a **variable** or a **function**
- Variables are containers for storing data values
 - Defined using a data type
 - `int a`
 - `float w = 10.89`
- A function is a block of code which only runs when it is called
 - Defined using function name, return type and arguments

Variables

- A variable is a name of the memory location
- It is used to store data
- Its value can be changed, and it can be reused many times
- Declaration of variables require data type to be specified

Variable Data Types

- Each variable in C has an associated data type
- Each data type requires different amounts of memory and has some specific operations which can be performed over it



Variable Data Types – Range & Size

Data Type	Range	Bytes	Format
signed char	-128 to + 127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf
Note: The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 16-bit compiler.			

Variable Data Type - Example

```
#include <stdio.h>
```

```
void main(void) {
```

```
    int val = 32760; // maximum signed is 32767
```

```
    printf("val+10 is %d\n", val+10);
```

```
}
```

Variable Definition

Variable Declaration

Format Specifier

This code will print -32766. Note that $32760 + 10 = 32770$. If you subtract 2^{16} from this quantity you obtain the wrapped-around result, -32766.

Data Input/Output - integer

```
#include <stdio.h>

void main()
{
    int a, b, c;
    printf("Please enter any two numbers: \n");
    scanf("%d %d", &a, &b);
    c = a + b;
    printf("The addition of two number is: %d", c);
}
```

Output

```
Please enter any two numbers:
12
3
The addition of two number is:15
```

Data Input/Output – int, float, char

```
#include <stdio.h>
int main()
{
    int Integer;
    char Character;
    float InputFloat;

    printf(" Please Enter a Character : ");
    scanf("%c", &Character);

    printf(" Please Enter an Integer Value : ");
    scanf("%d", &Integer);

    printf(" Please Enter Float Value : ");
    scanf("%f", &InputFloat);

    printf(" \n The Integer Value that you Entered is : %d", Integer);
    printf(" \n The Character that you Entered is : %c", Character);
    printf(" \n The Float Value that you Entered is : %f", InputFloat);
    printf(" \n The Float Value with precision 2 is : %.2f", InputFloat);

    return 0;
}
```

Output

```
Please Enter a Character :  s
Please Enter an Integer Value :  125
Please Enter Float Value :  16.568975

The Integer Value that you Entered is :  125
The Character that you Entered is :  s
The Float Value that you Entered is :  16.568975
The Float Value with precision 2 is :  16.57
```

Example command	Output	Explanation of the format
<code>print \$format_double(".3f", 54.27305555)</code>	54.273	Print the integer part plus 3 decimal places
<code>print \$format_double(".3g", 54.27305555)</code>	54.3	Print 3 significant figures of the number
<code>print \$format_double(".1s", 54.27305555)</code>	54:16:23.0	Print in sexagesimal (base 60) with 1 decimal place

Data Input/Output – floating point

Data Input/Output – string

```
#include <stdio.h>

int main()
{
    // array to store string taken as input
    char color[20];

    // take user input
    printf("Enter your favourite color: ");
    scanf("%s", color);

    // printing the input value
    printf("Your favourite color is: %s.", color);

    return 0;
}
```

Output

```
Enter any sentence: Orange
Your favourite color is: Orange.
```

- %s in scanf() can be used to accept string inputs
- Name of the string is mentioned in the function as it points to the initial address of character array

Data Input/Output – string

```
#include <stdio.h>

int main()
{
    // array to store string taken as input
    char sentence[20];

    // take user input
    printf("Enter any sentence: ");
    scanf("%s", sentence);

    // printing the input value
    printf("You entered: %s.", sentence);

    return 0;
}
```

Output

```
Enter any sentence: This is my sentence
You entered: This.
```

- If the size of the array is not sufficient, it may lead to overflow

Data Input/Output – string

```
#include <stdio.h>

int main()
{
    // array to store string taken as input
    char sentence[20];

    // take user input
    printf("Enter any sentence: ");

    // use the fgets method specifying the size of the array as the max limit
    fgets(sentence, 20, stdin);

    // printing the input value
    printf("You entered: %s.", sentence);

    return 0;
}
```

- User can mention the upper limit of the string input that can be stored in the variable
- If the entered string size is less than the mentioned size, entered content is stored as such
- If the entered content is having size greater than the mentioned size, the function restricts the input and buffer overflow is prevented

Output

```
Enter any sentence: I am going to the park today
You entered: I am going to the p.
```


Typecast

- Converting one datatype into another is known as type casting or, type-conversion

```
#include <stdio.h>

main() {

    int sum = 17, count = 5;
    double mean;

    mean = sum / count;
    printf("Value of mean : %f\n", mean );
}
```

Output

```
Value of mean : 3.000000
```

```
#include <stdio.h>

main() {

    int sum = 17, count = 5;
    double mean;

    mean = (double) sum / count;
    printf("Value of mean : %f\n", mean );
}
```

Output

```
Value of mean : 3.400000
```

Scope

- In C programming, every variable is defined in scope
- Scope can be regarded as the section or region of a program where a variable has its existence
 - The variable cannot be used or accessed beyond that region
- In C programming, a variable declared within a function differs from a variable declared outside of a function

Scope

```
#include <stdio.h>

//global variable definition
int z;

int main ()
{
    //local variable definition and initialization
    int x,y;

    //actual initialization
    x = 20;
    y = 30;
    z = x + y;

    printf ("value of x = %d, y = %d and z = %d\n", x, y, z);

    return 0;
}
```

x, y → **Local Variable**

- Defined within a function
- Can be accessed only within the function where it is defined
- In this case it is main()
- Values should be initialised by the user

z → **Global Variable**

- Defined outside main
- Can be accessed by main and all user defined function within the program
- Can be accessed outside the program by using extern keyword
- Value will get initialised to 0

datatype	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

Storage Class

- A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program
- Storage classes in a C program
 - auto
 - register
 - static
 - extern

Storage Class

auto

The auto storage class is the default storage class for all local variables.

```
{  
    int mount;  
    auto int month;  
}
```

register

The register storage class is used to define local variables that should be stored in a register instead of RAM.

This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location)

```
{  
    register int miles;  
}
```

Storage Class

static

- The static storage class instructs the compiler to keep a local variable in existence during the **life-time of the program** instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.
- The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.
- In C programming, when static is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

Storage Class

static

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

    while(count-->0) {
        func();
    }

    return 0;
}

/* function definition */
void func( void ) {

    static int i = 5; /* local static variable */
    i++;

    printf("i is %d and count is %d\n", i, count);
}
```

Output

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

Storage Class

extern

- The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When 'extern' is used, the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.
- In a multiple files scenario, if there is a global variable or function to be used in other files, then extern will be used in another file to provide the reference of defined variable or function.

Storage Class

extern

First File: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();

main() {
    count = 5;
    write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
    printf("count is %d\n", count);
}
```

terminal

```
$gcc main.c support.c
```

Output

```
count is 5
```

Class	Name of Class	Place of Storage	Scope	Default Value	Lifetime
auto	Automatic	RAM	Local	Garbage Value	Within a function
extern	External	RAM	Global	Zero	Till the main program ends. One can declare it anywhere in a program.
static	Static	RAM	Local	Zero	Till the main program ends. It retains the available value between various function calls.
register	Register	Register	Local	Garbage Value	Within the function

Operators & Separators

- Assists programmers using I/O devices without certain U.S.—English characters

Token class	Tokens
Simple operators	<code>! % ^ & * - + =</code> <code>~ . < > / ?</code>
Compound assignment operators	<code>+= -= *= /= %=</code> <code><<= >>= &= ^= =</code>
Other compound operators	<code>-> ++ -- << >></code> <code><= >= == != && </code>
Separator characters	<code>() [] { } , ; : ...</code>
Alternate token spellings	<code><% %> <: :> %: %::%:</code>

Operators	Equivalent Punctuators
<code><%</code>	<code>{</code>
<code>%></code>	<code>}</code>
<code><:</code>	<code>[</code>
<code>:></code>	<code>]</code>
<code>%:</code>	<code>#</code>
<code>%::%:</code>	<code>##</code>

Header file `is0646.h` defines macros that expand to certain operators

Operators - Arithmetic

Operator	Name	Description	Example
+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	$x \% y$
++	Increment	Increases the value of a variable by 1	$++x$
--	Decrement	Decreases the value of a variable by 1	$--x$

Operators - Assignment

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Operators – Comparison

Operator	Name	Example
<code>==</code>	Equal to	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

Operator - Logical

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	<code>x < 5 && x < 10</code>
	Logical or	Returns true if one of the statements is true	<code>x < 5 x < 4</code>
!	Logical not	Reverse the result, returns false if the result is true	<code>!(x < 5 && x < 10)</code>

Operator - sizeof

Program

```
#include <stdio.h>

int main() {
    int myInt;
    float myFloat;
    double myDouble;
    char myChar;

    printf("%lu\n", sizeof(myInt));
    printf("%lu\n", sizeof(myFloat));
    printf("%lu\n", sizeof(myDouble));
    printf("%lu\n", sizeof(myChar));

    return 0;
}
```

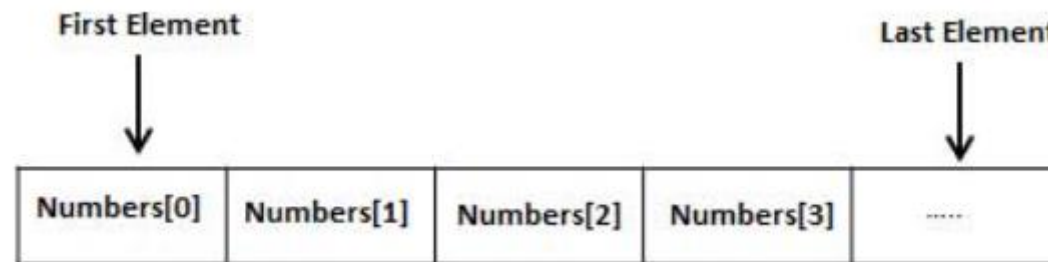
The memory size (in bytes) of a data type or a variable can be found with the sizeof operator

Output

```
4
4
8
1
```


Array

- Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type
- An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type



Syntax

```
type arrayName [ arraySize ];
```

Declaration

```
double balance[10];
```

Initialisation

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Defining a single element

```
balance[4] = 50.0;
```

Pointers

- Every variable is a memory location
- Every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory
- A pointer is a variable which stores memory address of another variable of same data type
- A pointer can be declared without data type using 'void'

```
#include <stdio.h>

int main () {

    int  var1;
    char var2[10];

    printf("Address of var1 variable: %x\n", &var1 );
    printf("Address of var2 variable: %x\n", &var2 );

    return 0;
}
```

Output

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

Pointer to a variable

```
#include <stdio.h>

void main()
{
    int a, *b;    // Defining an integer variable and a pointer
    b = &a;       // Assigning address of a to pointer b
    a = 45;       // Assigning value to a
    printf("Value of a: %d",a);    // Access using variable
    printf("\nValue of a: %d",*b); // Access using pointer
    printf("\nAddress of a: %x",&a); // Access using variable
    printf("\nAddress of a: %x",b); // Access using pointer
    printf("\nEnter a new value to a:");
    scanf("%d",b);
    printf("\nNew value of a: %d",*b);
}
```

Output:

```
Value of a: 45
Value of a: 45
Address of a: df9ff7a4
Address of a: df9ff7a4
Enter a new value to a:78

New value of a: 78
```

Void pointer?

NULL pointer?

Dangling pointer?

Conditional Statements

- Conditional statements require the usage of relational operators such as `>`, `<`, `>=`, `<=`, `==`
- Conditional statements offered in C
 - `if` to specify a block of code to be executed, if a specified condition is true
 - `else` to specify a block of code to be executed, if the same condition is false
 - `else if` to specify a new condition to test, if the first condition is false
 - `switch` to specify many alternative blocks of code to be executed

if Statement

```
if (condition) {  
    // block of code to be executed if the condition is true  
}
```

```
if (20 > 18) {  
    printf("20 is greater than 18");  
}
```

else Statement

```
if (condition) {  
    // block of code to be executed if the condition is true  
} else {  
    // block of code to be executed if the condition is false  
}
```

```
int time = 20;  
if (time < 18) {  
    printf("Good day.");  
} else {  
    printf("Good evening.");  
}
```

else if Statement

```
if (condition1) {  
    // block of code to be executed if condition1 is true  
}  
else if (condition2) {  
    // block of code to be executed if the condition1 is false and condition2 is true  
}  
else {  
    // block of code to be executed if the condition1 is false and condition2 is false  
}
```

```
int time = 22;  
if (time < 10) {  
    printf("Good morning.");  
} else if (time < 20) {  
    printf("Good day.");  
} else {  
    printf("Good evening.");  
}
```

Shorthand If...Else (Ternary Operator)

```
variable = (condition) ? expressionTrue : expressionFalse;
```

```
int time = 20;  
if (time < 18) {  
    printf("Good day.");  
} else {  
    printf("Good evening.");  
}
```



```
int time = 20;  
(time < 18) ? printf("Good day.") : printf("Good evening.");
```

Switch Statement

```
switch(expression) {  
    case x:  
        // code block  
        break;  
    case y:  
        // code block  
        break;  
    default:  
        // code block  
}
```

```
int day = 4;  
  
switch (day) {  
    case 1:  
        printf("Monday");  
        break;  
    case 2:  
        printf("Tuesday");  
        break;  
    case 3:  
        printf("Wednesday");  
        break;  
    case 4:  
        printf("Thursday");  
        break;  
    case 5:  
        printf("Friday");  
        break;  
    case 6:  
        printf("Saturday");  
        break;  
    case 7:  
        printf("Sunday");  
        break;  
}
```

```
int day = 4;  
  
switch (day) {  
    case 6:  
        printf("Today is Saturday");  
        break;  
    case 7:  
        printf("Today is Sunday");  
        break;  
    default:  
        printf("Looking forward to the Weekend");  
}
```


Looping Statements

- Loops can execute a block of code as long as a specified condition is reached
- Loops are handy because they save time, reduce errors, and they make code more readable

While Loop

```
while (condition) {  
    // code block to be executed  
}
```

```
int i = 0;  
  
while (i < 5) {  
    printf("%d\n", i);  
    i++;  
}
```

do/While Loop

```
do {  
    // code block to be executed  
}  
while (condition);
```

```
int i = 0;  
  
do {  
    printf("%d\n", i);  
    i++;  
}  
while (i < 5);
```

Output

```
0  
1  
2  
3  
4
```

for Loop

```
for (statement 1; statement 2; statement 3)  
    // code block to be executed  
}
```

```
int i;  
  
for (i = 0; i < 5; i++) {  
    printf("%d\n", i);  
}
```

Output

```
0  
1  
2  
3  
4
```

break and Continue

- **Break** was used to "jump out" of a switch statement/loop
- **Continue** statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop

```
int i;

for (i = 0; i < 10; i++)
{
    if (i == 4)
    {
        break;
    }
    printf("%d\n", i);
}

printf("\nThe Loop Ends");
```

Output

```
0
1
2
3
The Loop Ends
```

```
int i;

for (i = 0; i < 10; i++)
{
    if (i == 4)
    {
        continue;
    }
    printf("%d\n", i);
}

printf("\nThe Loop Ends");
```

Output

```
0
1
2
3
5
6
7
8
9
The Loop Ends
```

Array Operations

```
int myNumbers[] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++)
{
    printf("%d\n", myNumbers[i]);
}
```

```
int myNumbers[4];
int i;
for (i = 0; i < 3; i++)
{
    printf("Enter %d element: ", i);
    scanf("%d", &myNumbers[i]);
}
for (i = 0; i < 3; i++)
{
    printf("\nElement %d is: %d ", i, myNumbers[i]);
}
```

Printing Elements of an 1D Integer array

Output

```
25
50
75
100
```

Printing Elements of an 1D Integer array

Output

```
Enter 0 element: 12
Enter 1 element: 23
Enter 2 element: 45

Element 0 is: 12
Element 1 is: 23
Element 2 is: 45
```

Array Operations using Pointer

```
int myNumbers[] = {25, 50, 75, 100};
int i, *a;
a = myNumbers;
for (i = 0; i < 4; i++)
{
    printf("%d\n", *(a++));
}
```

Printing Elements of an 1D Integer array

Output

```
25
50
75
100
```

```
int myNumbers[4], *a;
a = myNumbers;
int i;
for (i = 0; i < 3; i++)
{
    printf("Enter %d element: ", i);
    scanf("%d", a+i);
}
for (i = 0; i < 3; i++)
{
    printf("\nElement %d is: %d ", i, *(a+i));
}
```

Printing Elements of an 1D Integer array

Output

```
Enter 0 element: 12
Enter 1 element: 23
Enter 2 element: 45

Element 0 is: 12
Element 1 is: 23
Element 2 is: 45
```

User Defined Functions

Function Declaration

```
returnType functionName(type1 parameterName1, type2 parameterName2, ...);
```

```
returnType functionName(type1 , type2, ...);
```

```
int getRectangleArea(int , int);
```

Function Definition

```
returnType functionName(functionParameters...)  
{  
    // function body  
}
```

```
int getRectangleArea(int length = 10, int breadth = 5) {  
    return length * breadth;  
}
```

Pass by Value

```
#include<stdio.h>

// function declaration
int getRectangleArea(int, int);

void main() {
    int l,b;
    printf("Enter length and breadth\n");
    scanf("%d %d", &l, &b);
    // function call
    int area = getRectangleArea(l, b);
    printf("Area of rectangle = %d", area);
}

// function definition
int getRectangleArea(int length, int breadth) {
    return length * breadth;
}
```

Output

```
Enter length and breadth
2 3
Area of rectangle = 6
```

Pass by Value

```
#include<stdio.h>

double getRatio(int numerator, int denominator) {
    // denominator is casted to double
    // to prevent integer division
    // result is casted to return type of function
    return (numerator / (double) denominator) ;
}

int main() {
    int a = 3, b = 2;

    double ratio = getRatio(a, b);
    printf("%d / %d = %.11f", a, b, ratio);

    return 0;
}
```

Output

```
3 / 2 = 1.5
```


Pass by Reference

```
#include<stdio.h>

void increment(int* a) {
    *a += 1;
}

int main() {
    int a = 5;
    printf("a before increment = %d\n", a);
    increment(&a); // call by reference
    printf("a after increment = %d\n", a);
    return 0;
}
```

Output

```
a before increment = 5
a after increment = 6
```

Function with No Return Value and without Argument

```
#include <stdio.h>
int fibo[10];
void generateFibo(); // function declaration

void main() {
    generateFibo(); // function call
    printf("First ten fibonacci numbers are :\n");
    int i;
    for (i = 0; i < 10 ; i++) {
        printf("%d, ", fibo[i]);
    }
}

void generateFibo() { // function definition
    fibo[0] = 1;
    fibo[1] = 1;
    int i;
    for (i = 2; i < 10 ; i++) {
        fibo[i] = fibo[i-1] + fibo[i-2];
    }
}
```

Output

```
First ten fibonacci numbers are :
1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
```

Function with No Return Value and with Arguments

```
#include <stdio.h>

// function declaration
void swap(int*, int*);

void main() {
    int a = 5, b = 6;
    printf("Before swap : a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After swap : a = %d, b = %d", a, b);
}

// function definition
void swap(int* a, int* b) {
    // function with no return value and with an argument
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Output

```
Before swap : a = 5, b = 6
After swap : a = 6, b = 5
```

Function with a Return Value and without Any Argument

```
#include <stdio.h>
#define PI 3.414

// function declaration
double circleArea();

int main() {
    double circle_area = circleArea(); // function call
    printf("Area of the circle = %0.2lf", circle_area);

    return 0;
}

// function definition
double circleArea() {
    // function with integer type return value and no argument
    int r;
    printf("Enter radius of the circle ");
    scanf("%d", &r);

    double area = PI * r * r;

    return area;
}
```

Output

```
Enter radius of the circle 7
Area of the circle = 167.29
```

Function with a Return Value and with an Argument

```
#include <stdio.h>
int isPrime(int); // function declaration
void main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    int is_number_prime = isPrime(number); // function call
    if (is_number_prime != 0) {
        printf("%d is a prime number", number);
    } else {
        printf("%d is not a prime number", number);
    }
}

int isPrime(int number) { // function definition
    if (number < 2) {
        return 0;
    }
    int i, result = 1;
    for (i = 2; i * i <= number; i++) {
        if (number % i == 0) {
            result = 0;
            break;
        }
    }
    return result;
}
```

Output

```
Enter a number: 5
5 is a prime number
```

Argument	Parameter
Also known as actual parameters	Also known as formal parameters
Arguments are used while calling the function	Parameters are used during the declaration of the function
Argument is the actual value of this variable that gets passed to function.	Parameter is variable in the declaration of the function.

Recursion

- Recursion: repetition of a process in a similar way until the specific condition reaches
- In C Programming, if a function calls itself from inside, the same function is called recursion
- The function which calls itself is called a recursive function, and the function call is termed a recursive call
- The recursion contains two cases in its program body.
 - **Base case:** When you write a recursive method or function, it keeps calling itself, so the base case is a specific condition in the function. When it is met, it terminates the recursion. It is used to make sure that the program will terminate. Otherwise, it goes into an infinite loop.
 - **Recursive case:** The part of code inside the recursive function executed repeatedly while calling the recursive function is known as the recursive case.

Recursion - Syntax

```
void recursive_fun() //recursive function
{
    Base_case; // Stopping Condition

    recursive_fun(); //recursive call
}

int main()
{
    recursive_fun(); //function call
}
```

```
recursive_function()
{
    //base case
    if base_case = true;
    return;

    else
    //recursive case
    return code_for_recursion; //includes recursive call
}
```


Direct Recursion

```
function_01()  
{  
    //some code  
    function_01();  
    //some code  
}
```

Output

```
Enter a digit for fibonacci series: 8  
0 1 1 2 3 5 8 13
```

```
#include<stdio.h>  
int fibonacci_01(int i)  
{  
    if (i == 0)  
        return 0;  
    if (i == 1)  
        return 1;  
    return fibonacci_01(i - 1) + fibonacci_01(i - 2);  
}  
  
int main()  
{  
    int i, n;  
    printf("Enter a digit for fibonacci series: ");  
    scanf("%d", & n);  
    for (i = 0; i < n; i++)  
        printf(" %d ", fibonacci_01(i));  
    return 0;  
}
```

Indirect Recursion

```
function_01()
{
    //some code
    function_02();
}

function_02()
{
    //some code
    function_01();
}
```

Output

```
2 1 4 3 6 5 8 7 10 9
```

```
#include<stdio.h>
int n=1;
void odd()
{
    if(n <= 10)
    {
        printf("%d ", n+1);
        n++;
        even();
    }
    return;
}

void even()
{
    if(n <= 10)
    {
        printf("%d ", n-1);
        n++;
        odd();
    }
    return;
}

int main()
{
    odd();
}
```

Calling a function from another file

file1.c

```
1  #include<stdio.h>
2  #include "file2.c"
3
4
5  void main()
6  {
7      int x, y, z;
8      x = 2;
9      y = 5;
10     z = 10;
11     printf("\n%d + %d + %d = %d", x, y, z, sum(x,y,z));
12     printf("\n%d * %d * %d = %d", x, y, z, mul(x,y,z));
13 }
```

file2.c

```
1  int sum(int a, int b, int c)
2  {
3      return a+b+c;
4  }
5
6  int mul(int a, int b, int c)
7  {
8      return a*b*c;
9  }
10
```

Output

```
2 + 5 + 10 = 17
2 * 5 * 10 = 100
```

You can include the entire source file in another source file using the `#include` directive. This is **not recommended** because it can lead to code duplication and maintenance issues.

Sharing Variables and Functions

Requirements

- “.c” file containing the shared function and variable
- “.c” file containing a main function calling the shared function and variable
- “.h” file containing the shared variable with extern storage class and function prototype of the shared function
- Compile the “.c” files together and execute in the terminal

example.h

```
// example.h
#ifndef EXAMPLE_H
#define EXAMPLE_H

// Declare a global variable
extern int shared_variable;

// Declare a function
void shared_function(void);

#endif // EXAMPLE_H
```

example.c

```
// example.c
#include "example.h" ←
#include <stdio.h>

// Define the global variable
int shared_variable = 42;

// Define the function
void shared_function(void) {
    printf("Shared function called! Variable value: %d\n", shared_variable);
}
```

Sharing Variables and Functions

main_fun.c

```
#include "example.h"
#include <stdio.h>

int main() {
    // Access the shared variable
    printf("Initial shared variable value: %d\n", shared_variable);

    // Modify the shared variable
    shared_variable = 100;

    // Call the shared function
    shared_function();

    return 0;
}
```

Terminal

```
PS C:\path> gcc -o my_program main_fun.c example.c
PS C:\path> ./my_program
Initial shared variable value: 42
Shared function called! Variable value: 100
```

