C Programming
Quick Reference

Geetesh Singh | Joses Paul | Kailash Sridharan
Lakshmanan | Malar Vizhi | Muthukumar | Satish Ram

ZILOGIC
SYSTEMS

C Programming

Quick Reference

Geetesh Singh | Joses Paul | Kailash Sridharan

Lakshmanan | Malar Vizhi | Muthukumar | Satish Ram

ZILOGIC
S Y S T E M S

Dedicated to Open source community

FOREWORD

This book 'C Quick Reference', tries to ease the complexity of reading ISO C Standard and gives a simpler sum-up of the standard with sample codes and diagrams. It demystifies some of the deep 'C' secrets and gives a greater insight on the design considerations of the language. It details the definitions, part of the standard, and parallels the cases which are unspecified and undefined in it.

This book is not only a good fit for beginners, but also becomes a handy reference for the seasoned professionals.

- Deepak Soundararajan

ACKNOWLEDGMENTS

CONTENTS

DATA TYPE SPECIFIERS

Data type specifiers in c are used to specify the type of data. In C it is necessary to tell the compiler what type of data the variable contains.

1.1 Data Types :
There are four main data types that are inbuilt in C they are,
>Int.
>Char.
>Float.
>Double.

Type Modifiers :
Even though everything comes under data type specifiers, the programmers call these type modifiers because it defines the amount of memory allocated to the variables by modifying the original data types. For example, even though the memory allocated for int is 2 bytes if it's a short int only one byte is allocated increasing the efficiency of memory allocation.

The four type modifiers are:
>Short.
>Long.
>Signed.
>Unsigned.

Others:
>void
>char
>_Bool
>_Complex
>struct-or-union-specifier
>enum-specifier

Type Conversion:

C is extremely flexible in handling the interaction of different data types. For example, with a few casts, you can easily multiply an unsigned character with a signed long integer, add it to a character pointer, and then pass the result to a function expecting a pointer to a structure. Programmers are used to this flexibility, so they tend to mix data types without worrying too much about what's going on behind the scenes.

To deal with this flexibility, when the compiler needs to convert an object of one type into another type, it performs what's known as a type conversion. There are two forms of type conversions: **explicit type conversions**, in which the programmer explicitly instructs the compiler to convert from one type to another by casting, and **implicit type conversions**, in which the compiler does "hidden" transformations of variables to make the program function as expected.

Conversion Rules

Integer Types: Value Preservation

An important concept in integer type conversions is the notion of a value-preserving conversion. Basically, if the new type can represent all possible values of the old type, the conversion is said to be value-preserving. In this situation, there's no way the value can be lost or changed as a result of the conversion. For example, if an unsigned char is converted into an int, the conversion is **value-preserving** because an int can represent all of the values of an unsigned char.

Example:

Assuming you're considering a two's complement machine, you know that an 8-bit unsigned char can represent any value between 0 and 255. You know that a 32-bit int can represent any value between -2147483648 and 2147483647. Therefore, there's no value the unsigned char can have that the int can't represent.

Correspondingly, in a **value-changing conversion**, the old type can contain values that can't be represented in the new type. For example, if you convert an int into an unsigned int, you have potentially created an intractable situation. The unsigned int, on a 32-bit machine, has a range of 0 to 4294967295, and the int has a

range of -2147483648 to 2147483647. The unsigned int can't hold any of the negative values a signed int can represent.

Integer Types: Widening

When you convert from a narrow type to a wider type, the machine typically copies the bit pattern from the old variable to the new variable, and then sets all the remaining high bits in the new variable to 0 or 1. If the source type is unsigned, the machine uses **zero extension**, in which it propagates the value 0 to all high bits in the new wider type. If the source type is signed, the machine uses **sign extension**, in which it propagates the sign bit from the source type to all unused bits in the destination type.

Integer Types: Narrowing

When converting from a wider type to a narrower type, the machine uses only one mechanism: truncation. The bits from the wider type that don't fit in the new narrower type are dropped. All narrowing conversions are value-changing because you're losing precision.

Value preserving conversion, Zero extension
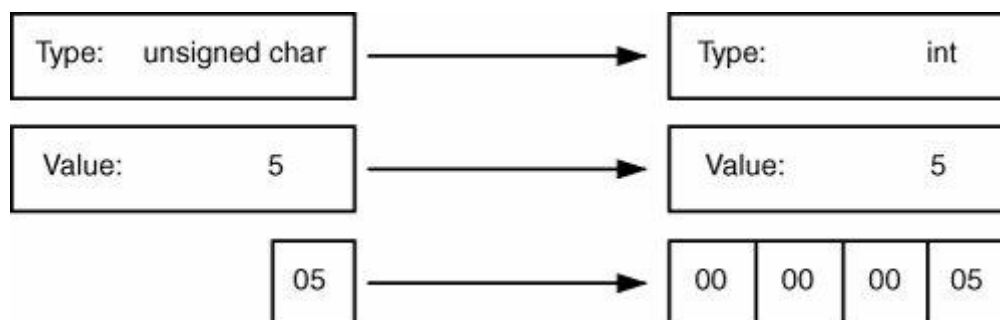
Type:	unsigned char	→	Type:	int

Value:	5	→	Value:	5

05	→	00	00	00	05

Value preserving conversion, Sign extension

Type:	signed char	→	Type:	int

Value:	-5	→	Value:	-5

FB	→	FF	FF	FF	FB

Value changing conversion, Sign extension

Type:	char	→	Type:	unsigned int

Value:	-5	→	Value:	4,294,967,291

FB	→	FF	FF	FF	FB
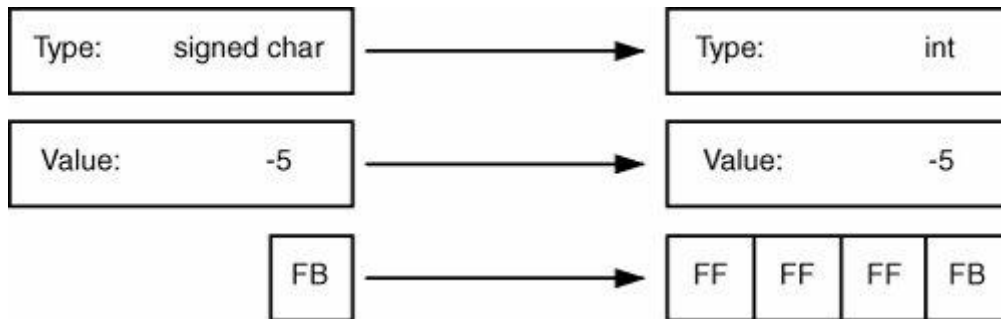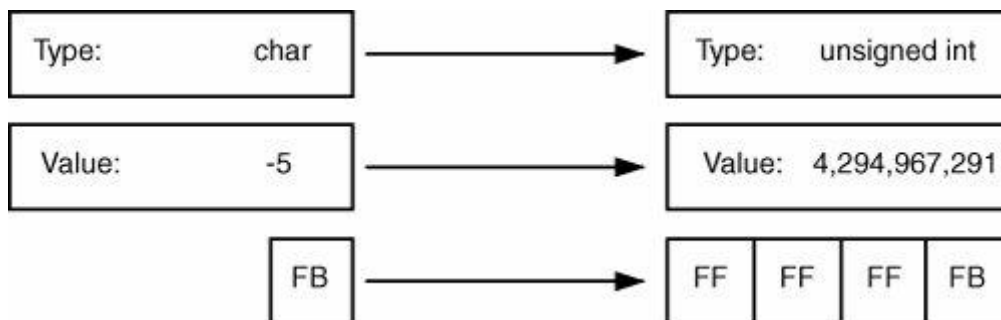
Value changing conversion, Truncation

Type:	int	→	Type:	unsigned short

Value:	-1000000	→	Value:	48576

FF	F0	BD	C0	→	BD	C0
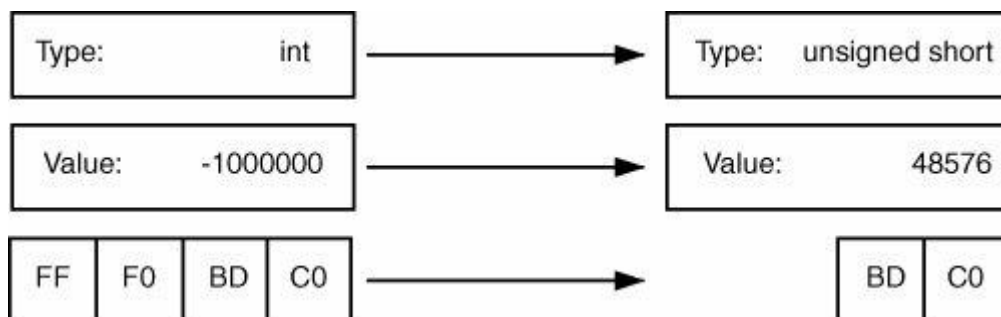
Integer Types: Signed and Unsigned

One final type of integer conversion to consider: If a conversion occurs between a signed type and an unsigned type of the same width, nothing is changed in the bit pattern. This conversion is value-changing.
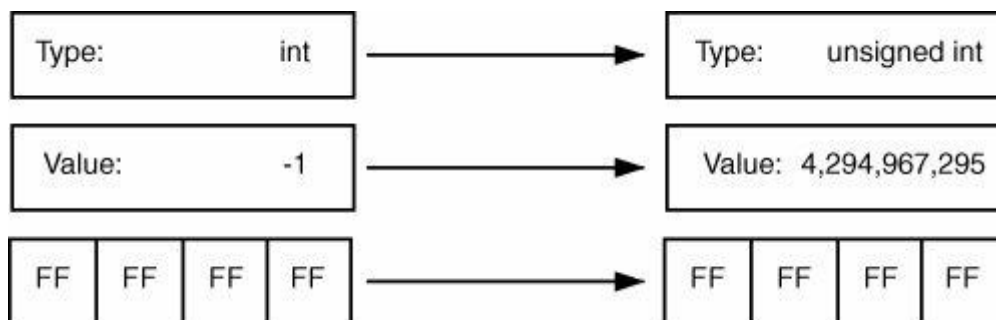
Floating Point and Complex Types

The C standard's rules for conversions between real floating types and integer types leave a lot of room for implementation-defined behaviors. In a conversion from a real type to an integer type, the fractional portion of the number is discarded. If the integer type can't represent the integer portion of the floating point number, the result is undefined. Similarly, a conversion from an integer type to a real type transfers the value over if possible. If the real type can't represent the integer's value but can come close, the compiler rounds the integer to the next highest or lowest number in an implementation-defined manner. If the integer is outside the range of the real type, the result is undefined.

Conversions between floating point types of different precision are handled with similar logic. Promotion causes no change in value. During a demotion that causes a change in value, the compiler is free to round numbers, if possible, in an implementation-defined manner. If rounding isn't possible because of the range of the target type, the result is undefined.

Based on three important concepts compiler chooses which conversion to apply in the context of C expressions:

- Simple conversions
- Integer promotions, and
- Usual arithmetic conversions.

1.Simple conversions

Simple conversions are C expressions that use straightforward applications of conversion rules.

Casts

As you know, typecasts are C's mechanism for letting programmers specify an explicit type conversion, as shown in this example:

(unsigned char) bob

Whatever type bob happens to be, this expression converts it into an unsigned char type. The resulting type of the expression is unsigned char.

Assignments

Simple type conversion also occurs in the assignment operator. The compiler must convert the type of the right operand into the type of the left operand, as shown in this example:

short int fred;

int bob = -10;

fred = bob;

For both assignments, the compiler must take the object in the right operand and convert it into the type of the left operand. The conversion rules tell you that conversion from the int bob to the short int fred results in truncation.

Function Calls: Prototypes

C has two styles of function declarations: the old K&R style, in which parameter types aren't specified in the function declaration, and the new ANSI style, in which the parameter types are part of the declaration. In the ANSI style, the use of function prototypes is still optional, but it's common.

With the ANSI style, you typically see something like this:

```
int dostuff (int jim, unsigned char bob);
void func(void)
{
  char a=42;
  unsigned short b=43;
  long long int c;
  c=dostuff(a, b);
}
```

The function declaration for dostuff() contains a prototype that tells the compiler the number of arguments and their types. The rule of thumb is that if the function has a prototype, the types are converted in a straightforward fashion using the rules documented previously. If the function doesn't have a prototype, something called the default argument promotions kicks in (explained in "Integer Promotions").

The previous example has a character (a) being converted into an int (jim), an unsigned short (b) being converted into an unsigned char (bob), and an int (the dostuff() function's return value) being converted into a long long int (c).

Function Calls: return

return does a conversion of its operand to the type specified in the enclosing function's definition. For example, the int a is converted into a char data type by **return**

```
char func(void) {
    int a=42;
    return a;
}
```

2.Integer Promotions

Integer promotions specify how C takes a narrow integer data type, such as a char or short, and converts it to an int (or, in rare cases, to an unsigned int). This up-conversion, or promotion, is used for two different purposes:

- Certain operators in C require an integer operand of type int or unsigned int. For these operators, C uses the integer promotion rules to transform a narrower integer operand into the correct typeint or unsigned int.
- Integer promotions are a critical component of C's rules for handling arithmetic expressions, which are called the usual arithmetic conversions. For arithmetic expressions involving integers, integer promotions are usually applied to both operands.

Each integer data type is assigned what's known as an integer conversion rank. These ranks order the integer data types by their width from lowest to highest. The signed and unsigned varieties of each type are assigned the same rank. The following abridged list sorts integer types by conversion rank from high to low.

- long long int, unsigned long long int
- long int, unsigned long int
- unsigned int, int
- unsigned short, short
- char, unsigned char, signed char
- _Bool

What happens when applying integer promotion?

First, if the variable isn't an integer type or a bit field, the promotions do nothing. Second, if the variable is an integer type, but its integer conversion rank is greater than or equal to that of an int, the promotions do nothing. Therefore, ints, unsigned ints, long ints, pointers, and floats don't get altered by the integer promotions.

So, the integer promotions are responsible for taking a narrower integer type or bit field and promoting it to an int or unsigned int. This is done in a straightforward fashion:

If a value-preserving transformation to an int can be performed, it's done. Otherwise, a value-preserving conversion to an unsigned int is performed. In practice, this means almost everything is converted to an int, as an int can hold the minimum and maximum values of all the smaller types.

Integer Promotion Applications

Unary + Operator

The unary + operator performs integer promotions on its operand. For example, if the bob variable is of type char, the resulting type of the expression (+bob) is int, whereas the resulting type of the expression (bob) is char.

Unary - Operator

The unary - operator does integer promotion on its operand and then does a negation. Regardless of whether the operand is signed after the promotion, a two's complement negation is performed, which involves inverting the bits and adding 1.

Unary ~ Operator

The unary ~ operator does a one's complement of its operand after doing an integer promotion of its operand. This effectively performs the same operation on both signed and unsigned operands for two's complement implementations: It inverts the bits.

Bitwise Shift Operators

The bitwise shift operators >> and << shift the bit patterns of variables. The integer promotions are applied to both arguments of these operators, and the type of the result is the same as the promoted type of the left operand, as shown in this

Example:

```
char a = 1;
char c = 16;
int bob;
bob = a << c;
```

a is converted to an integer, and c is converted to an integer. The promoted type of the left operand is int, so the type of the result is an int. The integer representation of a is left-shifted 16 times.

Switch Statements

Integer promotions are used in switch statements. The general form of a switch statement is something like this:

```
switch (controlling expression)
{
case (constant integer expression): body;
break;
default: body;
break;
}
```

The integer promotions are used in the following way: First, they are applied to the controlling expression, so that expression has a promoted type. Then, all the integer constants are converted to the type of the promoted control expression.

Function Invocations

Older C programs using the K&R semantics don't specify the data types of arguments in their function declarations. When a function is called without a prototype, the compiler has to do something called default argument promotions. Basically, integer promotions are applied to each function argument, and any arguments of the float type are converted to arguments of the double type. Consider the following example:

```
int jim(bob)
char bob;
{
printf("bob=%d\n", bob);
}


int main(int argc, char **argv)
{
char a=5;
jim(a);
}
```

In this example, a copy of the value of a is passed to the jim() function. The char type is first run through the integer promotions and transformed into an integer. This integer is what's passed to the jim() function. The code the compiler emits for the jim() function is expecting an integer argument, and it performs a direct conversion of that integer back into a char format for the bob variable.

3.Usual Arithmetic Conversions

In many situations, C is expected to take two operands of potentially divergent types and perform some arithmetic operation that involves both of them. The C standards spell out a general algorithm for reconciling two types into a compatible type

for this purpose. This procedure is known as the usual arithmetic conversions. The goal of these conversions is to transform both operands into a common real type, which is used for the actual operation and then as the type of the result. These conversions apply only to the arithmetic types integer and floating point types. The following sections tackle the conversion rules.

✓ **Rule 1: Floating Points Take Precedence**

Floating point types take precedence over integer types, so if one of the arguments in an arithmetic expression is a floating point type, the other argument is converted to a floating point type. If one floating point argument is less precise than the other, the less precise argument is promoted to the type of the more precise argument.

✓ **Rule 2: Apply Integer Promotions**

If you have two operands and neither is a float, you get into the rules for reconciling integers. First, integer promotions are performed on both operands. This is an extremely important piece of the puzzle! If you recall from the previous section, this means any integer type smaller than an int is converted into an int, and anything that's the same width as an int, larger than an int, or not an integer type is left alone. Here's a brief example:

```
unsigned char jim = 255;
unsigned char bob = 255;
if ((jim + bob) > 300) do_something();
```

In this expression, the + operator causes the usual arithmetic conversions to be applied to its operands. This means both jim and bob are promoted to ints, the addition takes place, and the resulting type of the expression is an int that holds the result of the addition (510). Therefore, do_something() is called, even though it looks like the addition could cause a numeric overflow. To summarize: Whenever there's

arithmetic involving types narrower than an integer, the narrow types are promoted to integers behind the scenes. Here's another brief example:

```
unsigned short a=1;
if ((a-5) < 0) do_something();
```

Intuition would suggest that if you have an unsigned short with the value 1, and it's subtracted by 5, it underflows around 0 and ends up containing a large value. However, if you test this fragment, you see that do_something() is called because both operands of the subtraction operator are converted to ints before the comparison. So a is converted from an unsigned short to an int, and then an int with a value of 5 is subtracted from it. The resulting value is -4, which is a valid integer value, so the comparison is true. Note that if you did the following, do_something() wouldn't be called:

```
unsigned short a=1;
a=a-5;
if (a < 0) do_something();
```

The integer promotion still occurs with the (a-5) , but the resulting integer value of -4 is placed back into the unsigned short a . As you know, this causes a simple conversion from signed int to unsigned short, which causes truncation to occur, and a ends up with a large positive value. Therefore, the comparison doesn't succeed.

✓ **Rule 3: Same Type After Integer Promotions**

If the two operands are of the same type after integer promotions are applied, you don't need any further conversions because the arithmetic should be straightforward to carry out at the machine level. This can happen if both operands have been promoted to an int by integer promotions, or if they just happen to be of the same type and weren't affected by integer promotions.

✓ **Rule 4: Same Sign, Different Types**

If the two operands have different types after integer promotions are applied, but they share the same signed-ness, the narrower type is converted to the type of the wider type. In other words, if both operands are signed or both operands are unsigned, the type with the lesser integer conversion rank is converted to the type of the operand with the higher conversion rank.

Note that this rule has nothing to do with short integers or characters because they have already been converted to integers by integer promotions. This rule is more applicable to arithmetic involving types of larger sizes, such as long long int or long int. Here's a brief example:

```
int jim =5;
long int bob = 6;
long long int fred;
fred = (jim + bob)
```

Integer promotions don't change any types because they are of equal or higher width than the int type. So this rule mandates that the int jim be converted into a long int before the addition occurs. The resulting type, a long int, is converted into a long long int by the assignment to fred.

✓ **Rule 5: Unsigned Type Wider Than or Same Width as Signed Type**

The first rule for this situation is that if the unsigned operand is of greater integer conversion rank than the signed operand, or their ranks are equal, you convert the signed operand to the type of the unsigned operand. This behavior can be surprising, as it leads to situations like this:

```
int jim = -5;
if (jim < sizeof (int))
do_something();
```

The comparison operator < causes the usual arithmetic conversions to be applied to both operands. Integer promotions are applied to jim and to sizeof(int) , but they don't affect them. Then you continue into the usual arithmetic conversions and attempt to figure out which type should be the common type for the comparison. In this case, jim is a signed integer, and sizeof(int) is a size_t , which is an unsigned integer type. Because size_t has a greater integer conversion rank, the unsigned type takes precedence by this rule. Therefore, jim is converted to an unsigned integer type, the comparison fails, and do_something() isn't called. On a 32-bit system, the actual comparison is as follows:

```
if (4294967291 < 4)
do_something();
```

✓ **Rule 6: Signed Type is Wider Than Unsigned Type, Value Preservation is Possible**

If the signed operand is of greater integer conversion rank than the unsigned operand, and a value-preserving conversion can be made from the unsigned integer type to the signed integer type, you choose to transform everything to the signed integer type, as in this example:

```
long long int a=10;
unsigned int b= 5;
(a+b);
```

The signed argument, a long long int, can represent all the values of the unsigned argument, an unsigned int, so the compiler would convert both operands to the signed operand's type: long long int.

✓ **Rule 7: Signed Type is Wider Than Unsigned Type, Value Preservation is Impossible**

There's one more rule: If the signed integer type has a greater integer conversion rank than the unsigned integer type, but all values of the unsigned integer

type can't be held in the signed integer type, you have to do something a little strange. You take the type of the signed integer type, convert it to its corresponding unsigned integer type, and then convert both operands to use that type.

Here's an example:

```
unsigned int a = 10;
long int b=20;
(a+b);
```

For the purpose of this example, assume that on this machine, the long int size has the same width as the int size. The addition operator causes the usual arithmetic conversions to be applied. Integer promotions are applied, but they don't change the types. The signed type (long int) is of higher rank than the unsigned type (unsigned int). The signed type (long int) can't hold all the values of the unsigned type (unsigned int), so you're left with the last rule. You take the type of the signed operand, which is a long int, convert it into its corresponding unsigned equivalent, unsigned long int, and then convert both operands to unsigned long int. The addition expression, therefore, has a resulting type of unsigned long int and a value of 30.

Enumeration constant (enum):
Syntax:
    Enum-specifier:
        enum identifier opt {enumerator-list}
        enum identifier opt {enumerator-list,}
        enum identifier

    Enumerator-list:
        Enumerator
        enumerator-list, enumerator

    enumerator:
        enumeration-constant
        enumeration-constant = constant-expression

An Enumeration is a list of constant integer values, as in

```
enum boolean { NO, YES };
```

The first name in the enum has value 0, the next 1 and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as in the below example:

```
enum months {
    JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };    /*FEB
= 2, MAR = 3 and so on…*/
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumerations as below example:

```
#include<stdio.h>
enum number {zero = 0, one, Zero = 0};
int main (void)
{
    printf("a:%d\nb:%d\nc:%d\n", a,b,c);
    return 0;
}
/*'zero' and 'Zero' has same value it's not a problem but you can't create an another
enum which have the same name in the number enum
Like: enum variable { zero = 0, x, y, z};
you can't use the same name 'zero' again in another enum in same program */
```

Example:

```
#include <stdio.h>
enum  escapes {  BELL  =  '\a',  BACKSPACE,  HORIZONTAL_TAB,  LINE_FEED,
VERTICAL_TAB, };
int main (void)
{
    printf("a%cb%c%cd%c", LINE_FEED, BACKSPACE, BELL, LINE_FEED);
    return 0;
}
/* In this program the ascii value of '\a' is  assigned to BELL in the enum and the values
continue the progression from the last specified ascii value.
*/
```

STORAGE CLASS SPECIFIERS

2.1. STORAGE CLASS

The storage class determines the part of memory where storage is allocated for an object (particularly variables and functions) and how long the storage allocation continues to exist.

➜ Tells the compiler how to store a variable.

➜ All variables defined in a C program get some physical location in memory where variable's value is stored. (Memory and CPU Register).

The storage class of a variable in C determines the following for a variable

1. Scope
2. Linkage
3. Storage duration (Lifetime)

2.1.1. SCOPE: *(C11 6.2.1)*

Scope describes the region or regions of a program that can access an identifier. A typical C variable has one of the following scopes:

- **Block scope** - A block is a region of code contained within an opening brace and the matching closing brace. For instance, the entire body of a function is a block. A variable defined inside a block has block scope, and it is visible from the point it is defined until the end of the block containing the definition.

- **Function scope** - It applies just to labels used with **goto** statements. This means that even if a label first appears inside an inner block in a function, its scope extends to the whole function. It would be confusing if you could use the same label inside two separate blocks, and function scope for labels prevents this from happening.

- **Function prototype scope** - Function prototype scope runs from the point the variable is defined to the end of the prototype declaration.

- **File scope** - A variable with its definition placed outside of any function has file scope. It is visible from the point it is defined to the end of the file containing the definition. file scope variables are also called global variables .

```c
#include <stdio.h>
int units = 0;                          /* a variable with file scope */
int mighty(int, int);                   /*Function prototype scope*/


int main(void)
{
units++;
mighty(2,3);
}


int mighty(int a,int b)
{
int c;                                  /*Block scope*/
if(c==0) {
printf("c=%d\n",a+b);
goto link;                              /*Function scope*/
}
{
link:
        printf("Hello\n");
}
}
```

2.1.2. Translation Units and Files

After C preprocessing stage the compiler sees a single file containing information from your source code file and all the header files. This single file is called a translation unit.

If your program consists of several source code files, then it will consist of several translation units, with each translation unit corresponding to a source code file and its included files.

2.1.3. LINKAGE: *(C11 6.2.2)*

A C variable has one of the following linkages:

- External linkage - File scope- can be used anywhere in a multifile program.
- Internal linkage - File scope - Single translation unit.
- No linkage - Variables with block scope, function scope, or function proto-
  type scope.

```c
int giants = 5;                          // file scope, external linkage
static int dodgers = 3;                  // file scope, internal linkage


func1()                                  //external linkage
{
}


int main()
{
int a=10;                                // Block scope, No linkage
extern int b;                            // Block scope, external linkage
}
```

2.1.4. STORAGE DURATION :   *(C11 6.2.4)*

Scope and linkage describe the visibility of identifiers. An object has a storage duration that determines its lifetime. A typical C object has one of the following four storage durations:

- **Static storage duration -** If an object has static storage duration, it exists throughout program execution. Variables with file scope have static storage duration. Note that for file scope variables, the keyword **static** indicates the linkage type, not the storage duration. A file scope variable declared using **static** has internal linkage, but all file scope variables, using internal linkage or external linkage, have static storage duration.

- **Thread storage duration -** Thread storage duration comes into play in concurrent programming, in which program execution can be divided into multiple threads. An object with thread storage duration exists from when it's declared until the thread terminates. Such an object is created when a declaration that would otherwise create a file scope object is modified with the keyword **_Thread_local** . When a variable is declared with this specifier, each thread gets its own private copy of that variable.

- **Automatic storage duration -** Variables with block scope normally have automatic storage duration. These variables have memory allocated for them when the program enters the block in which they are defined, and the memory is freed when the block is exited. The memory used for automatic variables can be reused. For example, after a function call terminates, the memory it used for its variables can be used to hold variables for the next function that is called.

➢ It is possible, however, for a variable to have block scope but static storage duration. To create such a variable, declare it inside a block and add the keyword **static** to the declaration:

```
#include<stdio.h>
int a;                          //File scope,External linkage,Static storage duration
static int b;                   //File scope,Internal linkage,Static storage duration

void more (int number)
{
int index;                      // Block scope, Automatic duration
static int ct = 0;              // Block scope, Static duration
return 0;
}
```

➢ Here the variable ct is stored in static memory; it exists from the time the program is loaded until the program terminates. But its scope is confined to the more() function block.

● **Allocated storage duration** -The object is allocated and deallocated per request by using dynamic memory allocation functions (malloc,calloc,realloc,free). The duration of the object is decided by the programmer.

2.2. Storage-class-specifier: *(C11 6.7.1)*

The storage class specifiers are

typedef

extern

static

_Thread_local

auto

register

typedef, _Thread_local are maybe discussed later and now only discussing about extern, static, auto, register.

Constraints

> At most, one storage-class specifier may be given in the declaration specifiers in a declaration, except that _Thread_local may appear with static or extern.

> In the declaration of an object with block scope, if the declaration specifiers include _Thread_local, they shall also include either static or extern. If _Thread_local appears in any declaration of an object, it shall be present in every declaration of that object.

> _Thread_local shall not appear in the declaration specifiers of a function declaration.

1.2.1 AUTO

A variable belonging to the automatic storage class has automatic storage duration, block scope, and no linkage. By default, any variable declared in a block or function header belongs to the automatic storage class.

```c
int loop(int n)
 {
   int m;                                    // m in scope
   scanf("%d", &m);
   {
     int i;                                  // both m and i in scope
     for (i = m; i < n; i++)
     puts("i is local to a sub-block\n");
   }
return m;                                    // m in scope, i gone
 }
```

Undefined behavior :

If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.When the value of an object is used while it is indeterminate the behavior is undefined.

2.2.2 STATIC

Variables with file scope have static storage duration. Static keyword with file scope variable indicates the internal linkage. Static keyword with auto variable indicates **Scope** of the variable is within the function or block but **lifetime** is throughout the program.

Unspecified behavior :

All objects with static storage duration shall be initialized (set to their initial values) before program startup. The manner and timing of such initialization are otherwise unspecified.

Undefined behavior :

The same identifier has both internal and external linkage in the same translation unit.

```
#include<stdio.h>
int a;
static int a;
main()
...
```

2.2.3 REGISTER

Variables are normally stored in computer memory. Register variables are stored in the CPU registers or, more generally, in the fastest memory available, where they can be accessed and manipulated more rapidly than regular variables.

Because a register variable may be in a register rather than in memory, you can't take the address of a register variable. In most other respects, register variables are the same as automatic variables.

It is not compulsory for the compiler to store the data in the RAM , depending on the availability and feasibility the compiler will decide whether to store the variable or not.

Register variables are used inside the functions or blocks only. **Scope** of the Register variable is within the function or block and **lifetime** is within the function or block. No **Linkage**.

That is, they have block scope, no linkage, and automatic storage duration. A variable is declared by using the storage class specifier **register**

```
int main(void)
{
register int quick;
…..
}
```

Note:
- Since Register keyword is only a suggestion one can never be sure if the variable will be stored in register or not.
- The maximum length of the variable must be the length of the register (i.e) if the register size is 64 bit the maximum size must be 64 bit.

2.2.4. EXTERN

DEFINITION-:

The **extern** keyword means "declare without defining". In other words, it is a way to explicitly declare a variable, or to force a declaration without a definition. It is

also possible to explicitly define a variable, i.e. to force a definition.It is done by assigning an initialization value to a variable.

In the C programming language, an external variable is a variable defined outside any function block. On the other hand, a local (automatic) variable is a variable defined inside a function block.

HOW ?

To understand how external variables relate to the extern keyword, it is necessary to know the difference between *defining* and *declaring* a variable.

When a variable is *defined*, the compiler allocates memory for that variable and possibly also initializes its contents to some value.

When a variable is *declared*, the compiler requires that the variable be defined elsewhere. The declaration informs the compiler that a variable by that name and type exists, but the compiler does not need to allocate memory for it since it is allocated elsewhere.

The extern keyword means "declare without defining". In other words, it is a way to explicitly declare a variable, or to force a declaration without a definition. It is also possible to explicitly define a variable, i.e. to force a definition. It is done by assigning an initialization value to a variable.

To said simply, while using the extern keyword for a variable, we are telling the compiler to don't allocate the memory for this variable, because this variable is present in the memory somewhere else.

If neither the extern keyword nor an initialization value are present, the statement can be either a declaration or a definition. It is up to the compiler to analyse the modules of the program and decide.

Extern variables are used inside the functions or outside the functions. **Scope** of the Extern variable is for all translation unit and **lifetime** is throughout the program. **Linkage** is for all translation unit.

If you want to use global variable i of file1.c in file2.c, then below are the points to remember:

1. main function shouldn't be there in file2.c
2. now global variable i can be shared with file2.c by two ways:

   a) by declaring with extern keyword in file2.c i.e extern int i;

   b) by defining the variable i in a header file and including that header file in file2.c.

Example 1:-

File1.c

```
/* Variable defined here*/
int global_variable = 37;    /* Definition checked against declaration */
```

File2.c

```
#include <stdio.h>
extern int global_variable;              /*Declaration of global_variable*/
int main (void)
{
      global_variable++;
      printf("%d\n",a);
      return 0;
}
```

```
$ gcc -Wall file1.c file2.c -o file
$ ./file
38
```

Example 2:-

file1.c

```
/* Variable defined here*/

int global_variable = 37;    /* Definition checked against declaration */
```

extern.h

```
extern int global_variable;   /*Declaration of global_variable*/
```

file2.c

```c
#include <stdio.h>
#include "extern.h"
int main (void)
{
        global_variable++;
        printf("%d\n",a);
        return 0;
}
```

Terminal command:

```
$ gcc -Wall file1.c file2.c -o file
$ ./file
38
```

Undefined behavior :   *(C11, J2, P-563)*

An identifier with external linkage is used, but in the program there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier (C11 6.9).

Example. 1:-

In file1.c is An identifier with external linkage is used (extern int global_variable;). The external definition for the identifier(global_variable) is used in both the file2.c and file3.c files. So it is undefined.

file1.c

```
#include <stdio.h>
extern int global_variable;
int main (void)
{
------
------
}
```

file2.c

```
int global_variable;
```

file3.c

```
int global_variable;
```

SUMMARY

Keyword	Storage	Scope	Life time	Initial Value
AUTO	Memory	Local / Block Scope	Exists as long as Control remains in the block	Garbage
REGISTER	CPU Register	Local / Block Scope	Local to the block in which variable is declared.	Garbage
EXTERN	Memory	Global / File Scope	Exists as long as variable is running Retains value within the function	Zero
STATIC	Memory	File Scope/ Block Scope	Exists throughout the program	Zero

2.2.4 TYPEDEF

The typedef specifier introduce a new name for a type rather than reserving space for a variable and is called as storage class specifier only for syntactic convenience.. In some ways, a typedef is similar to macro text replacement—it doesn't introduce a new type, just a new name for a type, but there is a key difference explained later.

Typedef looks exactly like a variable declaration, Instead of the declaration saying "this name refers to a variable of the stated type," the typedef keyword doesn't create a variable, but causes the declaration to say "this name is a synonym for the stated type."

Typedef creates aliases for data types rather than new data types. You can typedef any type.

typedef int *IP;

Just write a declaration for a variable with the type you desire. Have the name of the variable be the name you want for the alias. Write the keyword 'typedef ' at the start, as shown above. A typedef name cannot be the same as another identifier in the same block.

Typedef with pointers:

```
typedef int *IP;                    /* IP: Pointer to int */
typedef int (*FP);              /* FP: Pointer to function returning int */
typedef int F(int);            /* F: Function with one int parameter, returning int */
IP ip;                         /* ip: Pointer to int */
IP fip();                      /* fip: function returning  pointer to int */
FP fp;                         /* fp: Pointer to function returning int */
F *fp2;      /* fp2: Pointer to a function taking an int parameter and returning an int */
```

Typedef names for structure:

```
Struct MyStruct {
   int data1;
   char data2;
};
typedef struct MyStruct newtype;
newtype a;                              /* struct MyStruct a; */
```

Typedef names for arrays:

```
typedef double A5[5];          /* A5: 5 element array of double*/
typedef int A[ ];              /* A: array of int*/
A5 a5;
A a;                           /* a: array of int*/
A *ap3[3];                     /* ap3: 3-element array of pointers to array of int*/
```

If a typedef name denotes a variable length array type, the length of the array is fixed at the time the typedef name is defined, not each time it is used:

```
{
      int n=10;
      typedef int Array[n];
      n=25;
      Array a;
}
```

Typedef names for Function type:
<div align="center">

typedef double Func();
</div>

Func becomes a synonym for function returning double with this declaration. Func can be used to declare pointers to function type, array of pointers to function type.

<div align="center">

Func *f_ptr, *f_array[];
</div>

Array of functions cannot be declared. It is invalid.

<div align="center">

Func f_array[10];
</div>

Func cannot be used to define functions.

<div align="center">

Func f1 (int x)
{
…..
}
</div>

#define

#define is called as the macro replacement in c. When a macro is defined the given identifier is replaced by the replacement during the preprocessor state.the syntax is

#define <text> <replacement>

Example program
```
#include <stdio.h>
#define max(a,b) ((a) > (b) ? a:b)

Int main()
{
Int r;
r=max(6,5);
printf("the maximum of two nos is: %s , &r);
}
```

The above program is a simple implementation of a macro to find the largest of two numbers.from the above program it is evident that the usage of macro greatly increases the readability of the program.

Difference between typedef and #define:

#define is a C-directive which is also used to define the aliases for various data types similar to typedef but with the following differences –

- You can extend a macro typename with other type specifiers, but not a typedef 'd typename. That is,

> **#define peach int**
> **unsigned peach i**;   /* works fine */
> **typedef int banana;**
> **unsigned banana i**;   /* Bzzzt! illegal */

  Combining type qualifier with typedef names is allowed.

- Second, a typedef 'd name provides the type for every declarator in a declaration.
> **#define int_ptr int ***
> **int_ptr chalk, cheese;**

After macro expansion, the second line effectively becomes:
> **int * chalk, cheese;**

This makes chalk and cheese as different as chutney and chives: chalk is a pointer-to-an-integer, while cheese is an integer. In contrast, a typedef like this:

> **typedef char * char_ptr;**
> **char_ptr Bentley, Rolls_Royce;**

declares both Bentley and Rolls_Royce to be the same. The name on the front is different, but they are both a pointer to a char.

- typedef is limited to giving symbolic names to types only whereas #define can be used to define alias for values as well, q., you can define 1 as ONE etc.

- typedef interpretation is performed by the compiler whereas #define statements are processed by the preprocessor.

2.3. FAQs:

2.3.1. EXTERN:

Q1. What does extern mean in a function declaration?   *(cfaq 1.11)*

Ans : Extern is significant only with data declarations. In function declarations, it can be used as a stylistic hint to indicate that the function's definition is probably in another source file, but there is no formal difference between

```
extern int f();
```

and

```
int f();
```

Q2. I have an extern array which is defined in one file, and used in another:

file1.c:	*file2.c:*
int array[ ] = {1, 2, 3};	extern int array[ ];

Why doesn't sizeof work on array in file2.c?   *(cfaq 1.24)*

Ans: An extern array of unspecified size is an incomplete type; you cannot apply sizeof to it. sizeof operates at compile time, and there is no way for it to learn the size of an array which is defined in another file.

You have three options:

1. Declare a companion variable, containing the size of the array, defined and initialized (with sizeof) in the same source file where the array is defined:

file1.c:	*file2.c:*
int array[ ] = {1, 2, 3};	extern int array[ ];
int arraysz = sizeof(array);	extern int arraysz;

2. #define a manifest constant for the size so that it can be used consistently in the definition and the extern declaration:

file1.h:
```
#defineARRAYSZ 3
extern intarray[ARRAYSZ];
```

file1.c:
```
#include "file1.h"
int array[ARRAYSZ];
```

file2.c:
```
#include "file1.h"
```

3. Use some sentinel value (typically 0, -1, or NULL) in the array's last element, so that code can determine the end without an explicit size indication:

file1.c:
```
int array[] = {1, 2, 3, -1};
```

file2.c:
```
extern int array[ ];
```

(Obviously, the choice will depend to some extent on whether the array was already being initialized; if it was, option 2 is poor.)

2.3.2. STATIC:

Q1. Do all declarations for the same static function or variable have to include the storage class static? *(cfaq 1.10)*

Ans: The language in the Standard does not quite require this (what's most important is that the first declaration contain static), but the rules are rather intricate, and are slightly different for functions than for data objects. (There has also been a lot of historical variation in this area.) Therefore, it's safest if static appears consistently in the definition and all declarations.

Q2. What am I allowed to assume about the initial values of variables and arrays which are not explicitly initialized?
If global variables start out as ``zero", is that good enough for null pointers and floating-point zeroes? *(cfaq 1.30)*

Ans: Uninitialized variables with static duration (that is, those declared outside of functions, and those declared with the storage class static), are guaranteed to start out as zero, just as if the programmer had typed ``= 0'' or ``= {0}''. Therefore, such variables are implicitly initialized to the null pointer if they are pointers, and to 0.0 if they are floating-point.

Variables with automatic duration (i.e. local variables without the static storage class) start out containing garbage, unless they are explicitly initialized. (Nothing useful can be predicted about the garbage.) If they do have initializers, they are initialized each time the function is called (or, for variables local to inner blocks, each time the block is entered at the top[footnote]).

These rules do apply to arrays and structures (termed aggregates); arrays and structures are considered ``variables'' as far as initialization is concerned. When an automatic array or structure has a partial initializer, the remainder is initialized to 0, just as for statics. [footnote] See also question (cfaq 1.31).

Finally, dynamically-allocated memory obtained with malloc and realloc is likely to contain garbage, and must be initialized by the calling program, as appropriate.

Memory obtained with calloc is all-bits-0, but this is not necessarily useful for pointer or floating-point values (see cfaq question 7.31, and section 5).

Q3. I have a function which accepts, and is supposed to initialize, a pointer:

```
void f (int *ip)

    {

     static int dummy = 5;

     ip = &dummy;

    }
```

But when I call it like this:
int *ip;
f(ip);
the pointer in the caller remains unchanged.   *(cfaq 4.8)*

Ans: Are you sure the function initialized what you thought it did? Remember that arguments in C are passed by value. In the code above, the called function alters only the passed copy of the pointer. To make it work as you expect, one fix is to pass the address of the pointer (the function ends up accepting a pointer-to-a-pointer; in this case, we're essentially simulating pass by reference):

```
    void f(ipp)
    int **ipp;
    {
            static int dummy = 5;
            *ipp = &dummy;
    }
```

```
...
int *ip;
f(&ip);
```

Another solution is to have the function return the pointer:

```
int *f()
{
        static int dummy = 5;
        return &dummy;
}
...
int *ip = f();
```

Type Qualifiers

Type qualifiers in C. *(C11 6.7.3)*
- const
- volatile
- restrict
- _atomic.

VOLATILE

3.1.1 DEFINITION:
Volatile in C is a type qualifier that requests the compiler to suppress optimizations on the object.

3.1.2 HOW?
The purpose of volatile is to force an implementation to suppress optimization that could otherwise occur**.** Consider an embedded device with memory-mapped I/O, a pointer to its register can be qualified as volatile, in order to prevent compiler from optimizing, so that the register is accessed every time.

3.1.3 USES:
Volatile declaration may be used to describe an object corresponding to memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared shall not be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions. *(C11 6.7.3)*

3.1.4 SYNTAX:

<type qualifier> <type specifier> <variable name>

```
volatile int a;
```

3.1.5 UNDEFINED BEHAVIOR:
If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile type, behavior is undefined. *(C11 6.7.3)*

```
volatile int n = 1;  // object of volatile-qualified type
int* p = (int*)&n;
int val = *p;          //  undefined behavior
```

3.1.6 IMPLEMENTATION SPECIFIC:

There is no implementation-independent semantics for volatile object.   *(K&R A8.2)*

CONSTANT

3.2.1 DEFINITION:

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed (Which depends upon where const variables are stored, we may change value of const variable by using pointer).   The result is undefined if an attempt is made to change a const.

Key: A Constant object may be initialized but not thereafter assigned to.

3.2.2 HOW ?

Constant qualifier simply suggests compiler that programmer has no intention to change it, but other programs may or may not change that object.

3.2.3 SYNTAX:

<type qualifier> <type specifier> <variable name>

```
const int a;
```

3.2.4 UNDEFINED:

If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined.

(C11 6.7.3)

```
#include <stdio.h>
  int main()
 {
    const int a = 12;
    int *p;
    p = &a;
    *p = 70;
 }
```

- In this case, what will most often happen is that the answer will be "yes." A variable, const or not, is just a location in memory, and you can break the rules of constness and simply overwrite it. (Of course this will cause a severe bug if some other part of the program is depending on its const data being constant!)
- However in some cases -- most typically for const static data -- the compiler may put such variables in a read-only region of memory. MSVC, for example, usually puts const static ints in .text segment of the executable, which means that the operating system will throw a protection fault if you try to write to it, and the program will crash.
- In some other combination of compiler and machine, something entirely different may happen. The one thing you *can* predict for sure is that this pattern will annoy whoever has to read your code.

IMPLEMENTATION SPECIFIC:
The implementation may place a const object that is not volatile in a read-only region of storage. Moreover, the implementation need not allocate storage for such an object if its address is never used.

CONSTANT AND VOLATILE TOGETHER

An object marked as **const volatile** will not be permitted to be changed by the code (an error will be raised due to the **const** qualifier) - at least through that particular name / pointer.

The **volatile** part of the qualifier means that the compiler cannot optimize or reorder access to the object.

In an embedded system, this is typically used to access hardware registers that can be read and are updated by the hardware, but make no sense to write to (or might be an error to write to).

An example might be the status register for a serial port. Various bits will indicate if a character is waiting to be read or if the transmit register is ready to accept a new character (ie., - it's empty). Each read of this status register could result in a different value depending on what else has occurred in the serial port hardware.

It makes no sense to write to the status register (depending on the particular hardware spec), but you need to make sure that each read of the register results in an actual read of the hardware - using a cached value from a previous read won't tell you about changes in the hardware state.

quick example:

```
unsigned int const volatile *status_reg;    // assume these are assigned to point to the
unsigned char const volatile *recv_reg;     // correct hardware addresses

#define UART_CHAR_READY 0x00000001

int get_next_char()
{
    while ((*status_reg & UART_CHAR_READY) == 0) {
      // do nothing but spin
    }
return *recv_reg;
}
```

If these pointers were not marked as being volatile, a couple problems might occur:-

- the while loop test might read the status register only once, since the compiler could assume that whatever it pointed to would never change (there's nothing in the while loop test or loop itself that could change it). If you entered the function when there was no character waiting in UART hardware, you might end up in an infinite loop that never stopped even when a character was received.
- the read of the receive register could be moved by the compiler to before the while loop - again because there's nothing in the function that indicates that *recv_reg is changed by the loop, there's no reason it can't be read before entering the loop.

The **volatile** qualifiers ensure that these optimizations are not performed by the compiler.

FAQs:

1. **What's the difference between**
   const MAXSIZE = 100; and   #define MAXSIZE 100.

 A preprocessor #define gives you a true compile-time constant. In C, const gives you a run-time object which you're not supposed to try to modify; ``const'' really means ``read only''

#define:
- operates at compile time
- consumes no memory (though this is not too important)
- can use in compile-time constant expression
- uses different syntax; can make mistake with ;
- can't create pointers to
- no type checking

const:
- operates at run time
- consumes memory (though this is not too important)
- can't use in compile-time constant expression
- uses consistent syntax
- can create pointers to
- does type checking

2. **I don't understand why I can't use const values in initializers & array dimension in**
   const int n = 5;
   int a[n];

The const qualifier really means ``read-only''; an object so qualified is a runtime object which cannot (normally) be assigned to. The value of a const-qualified object is therefore *not* a constant expression in the full sense of the term, and cannot be used for array dimensions, case labels, and the like. (C is unlike C++ in this regard.) When you need a true compile-time constant, use a preprocessor #define (or perhaps an enum).

Experimentals

VOLATILE

Consider these code samples:

```
//file1.c
#include<stdio.h>
#include<time.h>

int main(void){

  long int i,j;
  double on,off,out;

  on=clock();      // on ← the starting clock time.
  for (i=0;i<90000;i++)
  for (j=0;j<50000;j++);
  off=clock();     // off ← the ending clock time.

  out=(off-on)/CLOCKS_PER_SEC;
  printf("Total Time taken ----> %lf \n",out);
  return 0;
}
```

```
//file2.c
#include<stdio.h>
#include<time.h>

int main(void){

  volatile  long int i,j;   // volatile qualified type.
  double on,off,out;

  on=clock();      // on ← the starting clock time.
  for (i=0;i<90000;i++)
  for (j=0;j<50000;j++);
  off=clock();     // off ← the ending clock time.

  out=(off-on)/CLOCKS_PER_SEC;
  printf("Total Time taken ----> %lf \n",out);
  return 0;
}
```
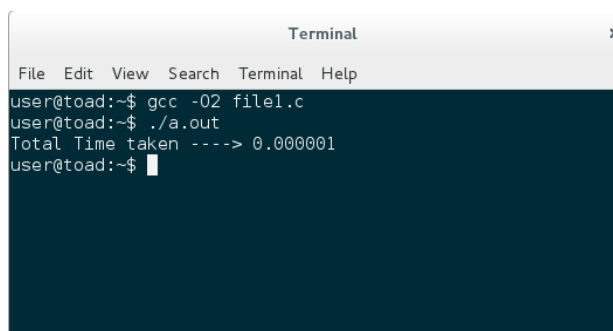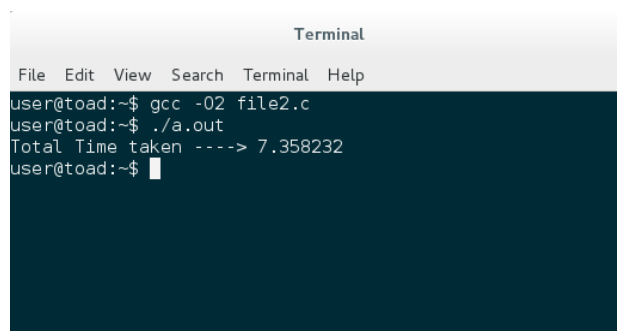
Note: Here, file2.c has a variable which is qualified as Volatile, now take a look at the output given by both the programs.

```
Terminal                                    ×
File  Edit  View  Search  Terminal  Help
user@toad:~$ gcc -O2 file1.c
user@toad:~$ ./a.out
Total Time taken ----> 0.000001
user@toad:~$ █
```

```
Terminal
File  Edit  View  Search  Terminal  Help
user@toad:~$ gcc -O2 file2.c
user@toad:~$ ./a.out
Total Time taken ----> 7.358232
user@toad:~$ █
```

It can be seen from the output that:
file1.c gives the output as "Total time taken is 0.000001". while file2.c which has the volatile qualified variable gives the output as "Total time taken is 7.358232". Although, the Gcc optimization level was specified as O2.

Perhaps, this is because of the fact that volatile suppresses the optimization, thus the loop gets executed.

FUNCTIONS

Functions: -

In C, A function provides a convenient way to encapsulate some computation, which can then be used without worrying about the implementation. C makes the use of functions easy, convenient and efficient.

We have used functions like **printf, scanf, etc.** These are the **C library** functions.

To develop own function, function definition has this form:(syntax)

```
return-type function-name(parameter declarations, if any)
{
    declarations
    statements
}
```

Return-type - what type of data is going to return by the function to the called one.

We will use *parameter* for a variable named in the parenthesized list in a function definition, and *argument* for the value used in a call of the function.

Why Functions?
- Maintenance
- Readability
- Size
- Boostspeed

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch.

Function Calls *(K&R A7.3.2)*

A function call is a postfix expression, called the function designator, followed by parentheses containing a possibly empty, comma-separated list of assignment expressions, which constitute the arguments to the function.

If the postfix expression consists of an identifier for which no declaration exists in the current scope, the identifier is implicitly declared as if the declaration had been given in the innermost block containing the function call.

<div align="center">

extern int identifier ( ) ;

</div>

The postfix expression must be of type "pointer to function returning T," :for some type T, and the value of the function call has type T.

The term **argument** is used for an expression passed by a function call; the term **parameter** is used for an input object (or its identifier) received by a function definition, or described in a function declaration. The terms "actual argument (parameter)" and "formal argument (parameter)" respectively are sometimes used for the same distinction.

```
int main(){
  multi(20);                                    // Argument or Actual
argument
}


void multi(int number) {                        // Parameter or Formal
argument
  printf("%d",number*number);
}
```

Constraints: *(c11, 6.5.2.2)*

1. The expression that denotes the called function shall have type pointer to function returning void or returning a complete object type other than an array type.

2. If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

Argument promotions:

If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type float are promoted to double. These are called the **default argument promotions.**

Function Declarators

In a new-style function declaration T D where D has the form

D 1 (parameter-type-list)

and the type of the identifier in the declaration T D1 is "type-modifier T," the type of the identifier of D is "type-modifier function with arguments parameter-type-list returning T."

In the new-style declaration, the parameter list specifies the types of the parameters. As a special case, the declarator for a new-style function with no parameters has a parameter type list consisting solely of the keyword **void**. If the parameter type list ends with an **ellipsis ", ..."**, then the function may accept more arguments than the number of parameters explicitly described.

The only storage-class specifier that shall occur in a parameter declaration is register.

what is ellipsis?

But first, let's identify what a va_list is and How to use variable argument lists (va_list) in C? Think about the printf() C function.

```
printf("Hello there! I like the numbers %d, %d and %d\n",1,3,7);
```

Obviously the output of that function call would be:

```
Hello there! I like the numbers 1, 3 and 7
```

But the key point here is, the printf() function can accept a VARYING NUMBER OF ARGUMENTS. That's because it uses a va_list.

If you look at the signature for printf(), it looks like this:

```
int printf(char * format, … );
```

So the argument list for printf() has 2 main things:

> char * format – a regular string
>
> and a second special argument, … (3 dots, just like that)

… is called an "ellipsis", and it means, in plain English: **"any number of optional arguments can go here."**

So somehow, in the innermost bowels of printf(), is some sticky code that somehow retrieves each one of the the list of args you're passing in, in the place of the "…".

Cool! So is it possible for us to write our functions that have their own sticky code that can process a set of VARIABLE ARGUMENTS???

YES YOU CAN. And it is actually simple!

Variable Argument Lists: < stdarg.h> *(K&R B7)*

The header <stdarg.h> provides facilities for stepping through a list of function arguments of unknown number and type.

Suppose *lastarg* is the last named parameter of a function **f** with a variable number of arguments. Then declare within **f** a variable ap of type **va_list** that will point to each argument in turn:

va_list ap;

ap must be initialized once with the macro **va_start** before any unnamed argument is accessed:

va_start(va_list ap, lastarg);

Thereafter, each execution of the macro **va_arg** will produce a value that has the type and value of the next unnamed argument, and will also modify ap so the next use of va_arg returns the next argument:

type va_arg (va_list ap, type);

The macro

void va_end(va_list ap);

must be called once after the arguments have been processed but before f is exited.

RULES you must know in order to be able to use "..." in one of your own functions:

1) The ... MUST appear exactly as ... It cannot be "..." (with the quotes), '...', or anything else weird.

2) The ... __MUST go last__ in the ARGUMENT LIST

3) There MUST be at least one mandatory, Non-optional argument, that comes before the ...

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

int addThemAll( int numargs, ... )
     // So this function can accept a variable number of arguments.  No limits.
{
    va_list listPointer;
    // POINTER that will be used to point first element of the VARIABLE ARGUMENT LIST.

     va_start( listPointer, numargs );
     // Make listPointer point to the first argument in the list
     // Notice that numargs is the LAST MANDATORY ARGUMENT
     // NEXT, we're going to start to actually retrieve the values from the va_list itself.

    int i;
    int sum = 0;
    for( i = 0 ; i < numargs; i++ )
        {
    // GET an arg.  YOU MUST KNOW  THE TYPE OF THE ARG TO RETRIEVE  IT FROM THE
va_list.
            int arg = va_arg( listPointer, int );
            printf( "    The %dth arg is %d\n", i, arg );
            sum += arg;
        }

    printf("--");
    printf("END OF ARGUMENT LIST\n\n");

    // FINALLY, we clean up by saying  va_end(). Don't forget to do this
    // BEFORE the addThemAll() function returns!
```

```
    va_end( listPointer );
     printf("The total sum was %d\n\n", sum);
    return sum;
}
int main()
{
     printf("Calling 'addThemAll( 3, 104, 29, 46 );' . . .\n");
    addThemAll( 3, 104, 29, 46 );

    printf("Calling 'addThemAll( 8,   1, 2, 3, 4, 5, 6, 7, 8 );' . . .\n");
    addThemAll( 8,   1, 2, 3, 4, 5, 6, 7, 8 );
    return 0;
}
```

Function Definition   *(c11, 6.9.1)*

Syntax

> *function-definition:*
>> *declaration-specifiers declarator declaration-list $_{opt}$ compound-statement*
>
> *declaration-list:*
>> *declaration*
>> *declaration-list declaration*

A function definition specifies the name of the function, the types and number of parameters it expects to receive, and its return type. A function definition also includes a function body with the declarations of its local variables, and the statements that determine what the function does.

The only storage-class specifiers that can modify a function declaration are **extern** and **static**. The **extern** specifier signifies that the function can be referenced from other files; that is, the function name is exported to the linker. The **static** specifier signifies that the function cannot be referenced from other files; that is, the name is not exported by the linker. If no storage class appears in a function definition, **extern** is assumed. In any case, the function is always visible from the definition point to the end of the file.

```
extern int max(int a, int b)
{
return a > b ? a : b;
}
```

extern is the storage-class specifier and **int** is the type specifier; **max(int a, int b)** is the function declarator; and **{ return a > b ? a : b; }** is the function body.

The following similar definition uses the identifier-list form for the parameter declarations:

```
extern int max(a, b)
int a, b;
{
return a > b ? a : b;
}
```

Here int a, b; is the declaration list for the parameters.

The return statement:

Constraints   *(c11, 6.8.6.4)*

1. A return statement with an expression shall not appear in a function whose return type is void. A return statement without an expression shall only appear in a function whose return type is void.

Semantics

1. A return statement terminates execution of the current function and returns control to its caller. A function may have any number of return statements.

2. If a return statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.

Function Pointer

- Pointer used to store address of function or pointer that points to function.
- Address of function is address of first instruction in the function.
- We can use a pointer to function to dereference and execute the function.

Example:

```
#include<stdio.h>
int Add(int a,int b)
{
   return a+b;
}
int main()
{
   int c;
   int (*p) (int,int);                        // Declaration of pointer
   p=&Add;
   c=(*p)(2,3);                               // Dereferencing and executing function
   printf("%d\n",c);
}
```

- In this example p is a pointer to a function that takes two int arguments and returns an integer.
- Function type and parameter list type should be same for the pointer.
- Instead of giving p=&Add you can give p=Add.
- p(2,3) does the same function as (*p)(2,3).

```c
#include<stdio.h>
void PrintHello()
{
    printf("Hello\n");
}
int main()
{
    void (*ptr) ();                          // Declaration of pointer
    ptr=PrintHello;
    ptr();
}
```

- ptr is pointer to function that returns nothing and no parameters.
- The char should be passed as pointer.

```c
#include<stdio.h>
void PrintHello(char *name)
{
    printf("Hello %s\n",name);
}
int main()
{
    void (*ptr) (*char);                     // Declaration of pointer
    ptr=PrintHello;
    ptr("Tom");
}
```

Use cases of Function pointer:

- Function pointers can be passed as arguments to function.
- Function would receive function pointer as argument can callback function that pointer will point to.

```
#include<stdio.h>
void A()
{
    printf("Hello\n");
}
void B(void (*ptr)())                           // Function pointer as argument
{
    ptr();                                       // Callback function that ptr points to
}
int main()
{
    void (*p) ()=A;                              // Declaration of pointer
    B(p);
}
```

Undefined Behaviour:

1. If the number of arguments does not equal the number of parameters, the behavior is undefined.

   (c11, 6.5.2.2, p6)

```
#include <stdio.h>
int main (void)
{

    fun1(5,76,79);

    return 0;
```

```
}
void fun1(int a,int b,int c,int d)
{
    printf("%d\n%d\n%d\n%d\n",a,b,c,d);
}
```

2. If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (, ...) --or the types of the arguments after**-promotion are not compatible with the types of the parameters, the behavior is undefined.

3. If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases:

   - one promoted type is a signed integer type, the other promoted type is the corresponding unsigned integer type, and the value is representable in both types;

   - both types are pointers to qualified or unqualified versions of a character type or void.

4. Two declarations of the same object or function specify types that are not compatible.

Statements

DEFINITION:

A statement specifies an action to be performed.

Different Statements & blocks in C are. *(C11 6.8)*
- labeled-statement
- compound-statement
- expression-statement
- selection-statement
- iteration-statement
- jump-statement

; → Statement terminator. *(K&R 3.1)*

Labeled Statement

SYNTAX:

identifier : statement
case constant-expression : statement
default : statement

Any statement (but not a declaration) may be preceded by any number of *labels*, each of which declares identifier to be a label name, which must be unique within the enclosing function (in other words, label names have function scope).

Example:

```
switch( Grade ) {
        case 'A' : printf( "Excellent\n" ); break;
        case 'B' : printf( "Good\n" ); break;
        case 'C' : printf( "OK\n" ); break;
        case 'D' : printf( "You must do better than this\n" ); break;
        default  : printf( "What is your grade anyway?\n" ); break;
    }
```

Compound Statement

A compound statement, or *block*, is a brace-enclosed sequence of statements and declarations.

SYNTAX:

{ statement | declaration...(optional) }

```
if (expr)                          // start of if-statement
{                                  // start of block
    int n = 1;                     // declaration
    printf("%d\n", n);             // expression statement
}                                  // end of block, end of if-statement
```

***Note**:- Each compound statement introduces its own block scope.*

```
int main(void)
{                                          // start of block
    {                                      // start of block
        puts("hello");                     // expression statement
        int n = printf("abc\n");           // declaration, prints "abc", stores 4 in n
        int a[n*printf("1\n")];            // allocates  8*sizeof(int)
        printf("%zu\n", sizeof(a));        // expression statement
    }                                      // end of block, scope of n and a ends
    int n = 7;                             // n can be reused
}
```

Expression Statement

SYNTAX:

expression(optional) ;

Most statements in a typical C program are expression statements, such as assignments or function calls.

```
puts("hello");          // expression statement
char *s;
while (*s++ != '\0')
    ;                   // null statement
```

Selection Statement

SYNTAX:

if (expression) statement
if (expression) statement else statement
switch (expression) statement

```
switch (expr)
{
     int i = 4;
     f(i);
     case 0:
          i = 17;            /* falls through into default code */
     default:
          printf("%d\n", i);
}
```

the object whose identifier is **i** exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the **printf() function** will access an indeterminate value. Similarly, the call to the **function f()** cannot be reached.

Iteration Statement

The iteration statements repeatedly execute a statement until condition false.

SYNTAX:

    while (expression **)** statement

    do statement **while (** expression **) ;**

    for (init_clause ; expression(optional) ; expression(optional)) statement

    for (expression$_{[opt]}$;expression$_{[opt]}$;expression$_{[opt]}$) statement.

Jump Statement

SYNTAX:

    break ;

    continue ;

    return expression(optional) ;

    goto identifier ;

```
for(i = 0; i < n; i++){
    for(j = 0; j < m; j++){
        if(a[i] == b[j]){
           goto found;          // Jumps to 'found' - out of all loops.
        else
          continue;
          }
      }
}
found:
return 0;
```

Experimentals

Switch-Case

Take a look at this code:

```c
#include <stdio.h>

void foo(void)
{
    printf("Function foo");
}

int main(void)
{
  float x = 1.1;
  switch (x)                                // Only int or char or enum.
  {
    int a=10;                               // a will not be initialized.
    printf("printing Ten - %d",a);          // will not be executed.
    fun();                                  // will not be executed.

    case 1.1: printf("Case 1.1 ???");       // only int or char or enum.
            break;
    default: printf("Choice other than 1, 2 and 3");
          break;
    case 1: printf("case 1");               // break ??
    case 2: printf("case 2");
    case 3: printf("case 3");
    case 2-1: printf("case 1 Again");       // duplicate case.
  }
  return 0;
}
```

Output:

- The expression used in switch must be integral type (int, char and enum). Any other type of expression is not allowed.
- Maximum Size of int that can be used is: sizeof(unsigned long int).
- default block can be placed anywhere.
- Remember to use break statement, if required.
- Two Case labels cannot be same. Error: duplicate case value.

Now, take a look at this code:

```c
#include <stdio.h>

int main(void)
{
        int value = 0;
        switch (value) {
        case 0:
                int i = 1;                      // i is defined here
                printf("value:%d\n",i);
        break;
        default:
```

```
                printf("Enter correct value\n");
                break;
        }
return 0;
}
```

Output:

Solution: put braces inside case.

```
        case 0: {
                    int i = 1;                    // i is defined here
                    printf("value:%d\n",i);
                }
                 break;
```

Why?
Case blocks can only contain statements.

```
int i =1;        // is not a statement.
```

```
{ int i =1;}     // is a compound statement.
```

Return statement

Take a look at this code:

```c
#include <stdio.h>

int welcome()
{
        return printf("welcome ");
}

int func()
{
        printf("zilogic systems\n");
        return printf("to all%d\n",welcome());
}

int main(void)
{
        printf("hello%d\n",func());
        return 0;
}
```
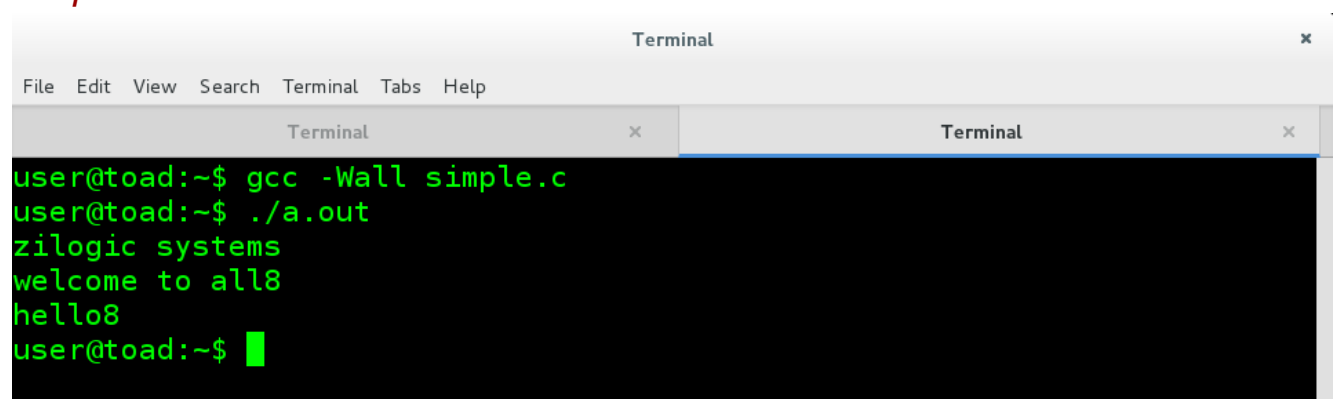
Output:

If - Conditional Statement

```c
#include <stdio.h>
int main(void){
        int condition = 0;
        if(condition)
                if(1)
                        printf("true");
         else
          printf("else");
         return 0;
}
```

Output:

```
                                    Terminal                              ✕
File  Edit  View  Search  Terminal  Tabs  Help
            Terminal              ×              Terminal                  ×
user@toad:~$ gcc -Wall simple.c
simple.c: In function 'main':
simple.c:6:11: warning: suggest explicit braces to avoid ambiguous 'else
' [-Wparentheses]
         if(condition)
           ^
user@toad:~$ ./a.out
user@toad:~$ █
```

Why?

Here, else is associated with the latest if condition.

Equivalent Code

```c
if(condition){
     if(1)
       printf("true");
      else
         printf("else");
}
```

Solution: Use Braces to associate the if and else, respective.

Structure

Origin: PASCAL - "Records"

Structure is a collection of one or more variables of same of different types, grouped under a same name. *(K&R Chapter 6)*

- A structure or union shall not contain a member with incomplete or function type (hence,a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself).*(C11 6.7.2.1 point no.3)*

WHY?

Organize complicated data.

SYNTAX:

Struct <struct_tag_name(opt)>{
        Member1;
        Member2;
        :
} < struct_variable_name(opt) >;
	Struct tag {
	    int x;
	    int y;
	    int z;
	 } variable1, variable2;

Initialization:

struct pinball {
   int right;
   int left;
   long score;
 }p1,p2;
 	
p1.right=40;
p1.left =60;
 	
p2.right=50;
p2.left=50;
	struct pinball {
	   int right;
	   int left;
	   long score;
	 };
	
	struct pinball
	p1={40,60,100};
	struct pinball
	p2={.score=100};
		struct pinball {
		   int right;
		   int left;
		   long score;
		}p1={40,60,100}

Size of Structure:

gcc on Linux - x64

```
Int main(void){
struct findsize{
   int a;
   char b;
   char c;
   int d;
 }var;

printf("size of var =
%d",sizeof(var));
return 0
}
```

a	a	a	a	b	c		
d	d	d	d				

```
$ gcc findsize.c

size of var = 12
```

Gcc – linux x64
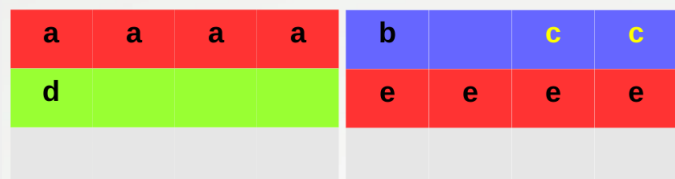
Int – 4
Char – 1
Short – 2
Long – 8
*ptr – 8

gcc on Linux - x64

```
int main(){
struct findsize{

int a;
char b;
short c;
char d;
int e;
}var;
printf("size of var =");
printf("%lu \n",sizeof(var));
```

a	a	a	a	b		c	c
d				e	e	e	e

```
$ gcc findsize.c

size of var = 12
```

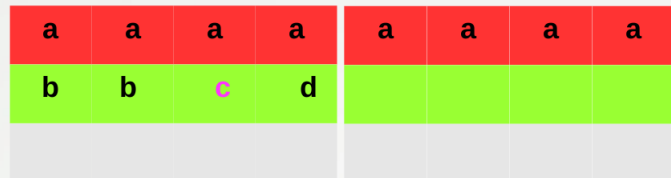Gcc – linux x64

Int – 4
Char – 1
Short – 2
Long – 8
*ptr – 8

> Generally it is prefered to club the types together inside a structure to minimise the wastage of memory.

Structure will tend allocate memory size of biggest member, to other members.It can be seen in the example below that, Since the structure has a member of type long. Short is given 8 bytes.Then the char c and d are put in the already available memory.

```
int main(){

struct findsize{
long a;
short b;
char c;
char d;
}var;

printf("size of var =");
printf("%lu \n",sizeof(var));
```

a	a	a	a	a	a	a	a
b	b	c	d				

$ gcc findsize.c

size of var = 16

Gcc – linux x64

Int – 4
Char – 1
Short – 2
Long – 8
*ptr – 8

Nested structure:-

```
struct  outer {
        int a;
        struct inner {
                long b;
         }var1;
}var2;
printf("%lu\n",sizeof(var2)); //size of the structure is 16 bytes in gcc
```

Ways to manipulate a structure. (using a function);
-    Pass elements of the structure.
-    Pass entire structure.
-    Pass a pointer to structure.

Pointer to a Structure

If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure. (K&R 6.2)

```
struct {                        // Structure.
    int a;
}var;

struct point *p = &var;     // pointer to the structure.
```

```
(*p).a
 p->a                         //accessing the member of the structure.
```

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

struct arith{
int a;
int b;
}one={1,1};

void add(struct arith *ptr){
int sum;
sum = (ptr->a) + (ptr->b);
printf("sum = %d",sum);
}

int main(){
struct arith *ptr = &one;
add(ptr);
return 0;
}
 /* Output : Sum = 2 */
```

FAQ:

Why is my compiler leaving holes in structures, wasting space and preventing ``binary'' I/O to external data files? Can I turn this off, or otherwise control the alignment of structure fields?

Many machines access values in memory most efficiently when the values are appropriately aligned. (For example, on a byte-addressed machine, short ints of size 2 might best be placed at even addresses, and long ints of size 4 at addresses which are a multiple of 4.) Some machines cannot perform unaligned accesses at all, and require that all data be appropriately aligned.
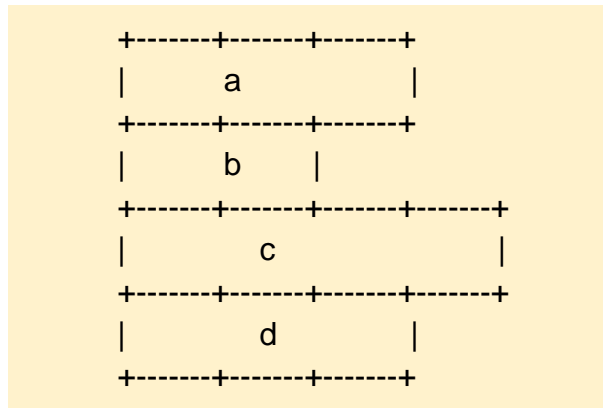
Suppose you have this structure:

```
struct {
        char a[3];
        short int b;
        long int c;
        char d[3];
        };
```
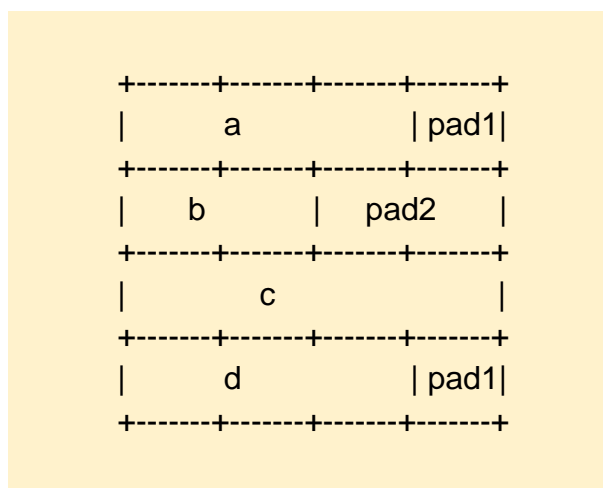
Now, you might think that it ought to be possible to pack this structure into memory like this:

```
+-------+-------+-------+-------+
|         a         | b   |
+-------+-------+-------+-------+
| b   |       c           |
+-------+-------+-------+-------+
| c   |       d           |
+-------+-------+-------+-------+
```

But it's much, much easier on the processor if the compiler arranges it like this:

```
+-------+-------+-------+
|       a       |
+-------+-------+-------+
|     b     |
+-------+-------+-------+-------+
|        c         |
+-------+-------+-------+-------+
|       d       |
+-------+-------+-------+
```

In the ``packed'' version, notice how it's at least a little bit hard for you and me to see how the b and c fields wrap around? In a nutshell, it's hard for the processor, too. Therefore, most compilers will ``pad'' the structure (as if with extra, invisible fields) like this:

```
+-------+-------+-------+-------+
|       a       | pad1|
+-------+-------+-------+-------+
|   b       |   pad2    |
+-------+-------+-------+-------+
|        c         |
+-------+-------+-------+-------+
|    d      | pad1|
+-------+-------+-------+-------+
```
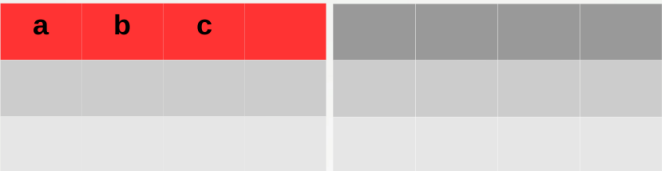
Bit Fields

When storage space is at premium, it may be necessary to pack several objects into a single machine word. Henceforth, comes the use of bit-fields.

gcc on Linux - x64

```
int main(){

struct findsize{
int a:8;
int b:8;
Int c:8;
}var;

printf("size of var =");
printf("%lu \n",sizeof(var));
```

| a | b | c |

```
$ gcc findsize.c

size of var = 4
```

Gcc – linux x64

Int – 4
Char – 1
Short – 2
Long – 8
*ptr – 8

- A member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a bit-field) its width is preceded by a colon. *(C11 6.7.2.1 point no.9)*

- The expression that specifies the width of a bit-field shall be an integer constant expression with a non-negative value that does not exceed the width of an object of the type that would be specified where the colon and expression omitted. *(C11 6.7.2.1 point no.4)*

- A bit-field shall have a type that is a qualified or unqualified version of _Bool, signed int, unsigned int, or some other implementation-defined type. *(C11 6.7.2.1 point no.5)*

Note:- The unary &(address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.

1) Multiple adjacent bit fields are permitted to be (and usually are) packed together:-

```
#include <stdio.h>
struct S {
        // will usually occupy 4 bytes:
        // 5 bits: value of b1
        // 11 bits: unused
        // 6 bits: value of b2
        // 2 bits: value of b3
        // 8 bits: unused
    unsigned b1 : 5, : 11, b2 : 6, b3 : 2;
};
int main(void)
{
    printf("%zu\n",sizeof(struct S)); // usually prints 4
}
```

2) The special *unnamed bit field* of width zero breaks up padding: it specifies that the next bit field begins at the beginning of the next allocation unit:-

```
#include <stdio.h>
struct S {
   // will usually occupy 8 bytes:
   // 5 bits: value of b1
   // 27 bits: unused
   // 6 bits: value of b2
   // 15 bits: value of b3
   // 11 bits: unused
   unsigned b1 : 5;
   unsigned :0; // start a new unsigned int
   unsigned b2 : 6;
   unsigned b3 : 15;
};
int main(void)
{
   printf("%zu\n", sizeof(struct S)); // usually prints 8
}
```

IMPLEMENTATION DEFINED:

- An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. *(C11 6.7.2.1 pt-11)*

C-FAQ:-

Why do people use explicit masks and bit-twiddling code so much, instead of declaring bit-fields?

- Bit-fields are thought to be nonportable, although they are no less portable than other parts of the language. (You don't know how big they can be, but that's equally true for values of type int. You don't know by default whether they're signed, but that's equally true of type char. You don't know whether they're laid out from left to right or right to left in memory, but that's equally true of the bytes of all types, and only matters if you're trying to conform to externally-imposed storage layouts, which is always nonportable.
- Bit-fields are inconvenient when you also want to be able to manipulate some collection of bits as a whole (perhaps to copy a set of flags). You can't have arrays of bit-fields. Many programmers suspect that the compiler won't generate good code for bit-fields (historically, this was sometimes true).

UNDEFINED:

- If the struct-declaration-list does not contain any named members, either directly or via an anonymous structure or anonymous union, the behavior is undefined. *(C11 6.7.2.1)*

Unions

- A union type describes an overlapping non empty set of member objects, each of which has an optionally specified name and possibly distinct type.
  (C11 6.2.5 pt.20)

- A union is a type consisting of a sequence of members whose storage overlaps (as opposed to struct, which is a type consisting of a sequence of members whose storage is allocated in an ordered sequence). The value of at most one of the members can be stored in a union at any one time.

Example:-

```
union pad {
        char c[5];
        float f;
} p = {.f = 1.23};
printf("size of union of char[5] and float is %zu\n", sizeof p);
```

OUTPUT: size of union of char[5] and float is 8

```
union S {
        uint32_t u32;
        uint16_t u16[2];
        uint8_t  u8;
        } s = {0x12345678};
        printf("Union S has size %zu and holds %x\n", sizeof s, s.u32);
        s.u16[0] = 0x0011;

//  printf("s.u8 is now %x\n", s.u8); // unspecified, typically 11 or 00
//  printf("s.u32 is now %x\n", s.u32); // unspecified, typically 12340011 or 00115678
```

OUTPUT :- Union S has size 4 and holds 12345678

UNSPECIFIED: -

When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values. *(C11 6.2.6.1-point no.7)*