# Chapter 4: Operators

An operator in a programming language is a symbol that tells the compiler or interpreter to perform a specific mathematical, relational or logical operation and produce a final result.

C has many powerful operators. Many C operators are binary operators, which means they have two operands. For example, in `a / b`, `/` is a binary operator that accepts two operands (`a`, `b`). There are some unary operators which take one operand (for example: `~`, `++`), and only one ternary operator `? :`.

## Section 4.1: Relational Operators

Relational operators check if a specific relation between two operands is true. The result is evaluated to 1 (which means *true*) or 0 (which means *false*). This result is often used to affect control flow (via `if`, `while`, `for`), but can also be stored in variables.

**Equals "=="**

Checks whether the supplied operands are equal.

```
1 == 0;         /* evaluates to 0. */
1 == 1;         /* evaluates to 1. */

int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr == yptr;   /* evaluates to 0, the operands hold different location addresses. */
*xptr == *yptr; /* evaluates to 1, the operands point at locations that hold the same value. */
```

Attention: This operator should not be confused with the assignment operator (`=`)!

**Not equals "!="**

Checks whether the supplied operands are not equal.

```
1 != 0;         /* evaluates to 1. */
1 != 1;         /* evaluates to 0. */

int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr != yptr;   /* evaluates to 1, the operands hold different location addresses. */
*xptr != *yptr; /* evaluates to 0, the operands point at locations that hold the same value. */
```

This operator effectively returns the opposite result to that of the equals (==) operator.

**Not "!"**

Check whether an object is equal to `0`.

The `!` can also be used directly with a variable as follows:

```
!someVal
```

This has the same effect as:

```
someVal == 0
```

**Greater than ">"**

Checks whether the left hand operand has a greater value than the right hand operand

```
5 > 4       /* evaluates to 1. */
4 > 5       /* evaluates to 0. */
4 > 4       /* evaluates to 0. */
```

**Less than "<"**

Checks whether the left hand operand has a smaller value than the right hand operand

```
5 < 4       /* evaluates to 0. */
4 < 5       /* evaluates to 1. */
4 < 4       /* evaluates to 0. */
```

**Greater than or equal ">="**

Checks whether the left hand operand has a greater or equal value to the right operand.

```
5 >= 4      /* evaluates to 1. */
4 >= 5      /* evaluates to 0. */
4 >= 4      /* evaluates to 1. */
```

**Less than or equal "<="**

Checks whether the left hand operand has a smaller or equal value to the right operand.

```
5 <= 4      /* evaluates to 0. */
4 <= 5      /* evaluates to 1. */
4 <= 4      /* evaluates to 1. */
```

# Section 4.2: Conditional Operator/Ternary Operator

Evaluates its first operand, and, if the resulting value is not equal to zero, evaluates its second operand. Otherwise, it evaluates its third operand, as shown in the following example:

```
a = b ? c : d;
```

is equivalent to:

```
if (b)
    a = c;
else
    a = d;
```

This pseudo-code represents it : `condition ? value_if_true : value_if_false`. Each value can be the result of an evaluated expression.

```
int x = 5;
int y = 42;
printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /* Outputs "5, 42" */
```

The conditional operator can be nested. For example, the following code determines the bigger of three numbers:

```
big= a > b ? (a > c ? a : c)
           : (b > c ? b : c);
```

The following example writes even integers to one file and odd integers to another file:

```
#include<stdio.h>

int main()
{
    FILE *even, *odds;
    int n = 10;
    size_t k = 0;

    even = fopen("even.txt", "w");
    odds = fopen("odds.txt", "w");

    for(k = 1; k < n + 1; k++)
    {
        k%2==0 ? fprintf(even, "\t%5d\n", k)
               : fprintf(odds, "\t%5d\n", k);
    }
    fclose(even);
    fclose(odds);

    return 0;
}
```

The conditional operator associates from right to left. Consider the following:

```
exp1 ? exp2 : exp3 ? exp4 : exp5
```

As the association is from right to left, the above expression is evaluated as

```
exp1 ? exp2 : ( exp3 ? exp4 : exp5 )
```

# Section 4.3: Bitwise Operators

Bitwise operators can be used to perform bit level operation on variables.
Below is a list of all six bitwise operators supported in C:

| Symbol | Operator |
|---|---|
| & | bitwise AND |
| \| | bitwise inclusive OR |
| ^ | bitwise exclusive OR (XOR) |
| ~ | bitwise not (one's complement) |
| << | logical left shift |
| >> | logical right shift |

Following program illustrates the use of all bitwise operators:

```
#include <stdio.h>

int main(void)
{
```

```
    unsigned int a = 29;    /* 29 = 0001 1101 */
    unsigned int b = 48;    /* 48 = 0011 0000 */
    int c = 0;

    c = a & b;              /* 32 = 0001 0000 */
    printf("%d & %d = %d\n", a, b, c );

    c = a | b;              /* 61 = 0011 1101 */
    printf("%d | %d = %d\n", a, b, c );

    c = a ^ b;              /* 45 = 0010 1101 */
    printf("%d ^ %d = %d\n", a, b, c );

    c = ~a;                 /* -30 = 1110 0010 */
    printf("~%d = %d\n", a, c );

    c = a << 2;             /* 116 = 0111 0100 */
    printf("%d << 2 = %d\n", a, c );

    c = a >> 2;             /* 7 = 0000 0111 */
    printf("%d >> 2 = %d\n", a, c );

    return 0;
}
```

Bitwise operations with signed types should be avoided because the sign bit of such a bit representation has a particular meaning. Particular restrictions apply to the shift operators:

- Left shifting a 1 bit into the signed bit is erroneous and leads to undefined behavior.
- Right shifting a negative value (with sign bit 1) is implementation defined and therefore not portable.
- If the value of the right operand of a shift operator is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

**Masking:**

Masking refers to the process of extracting the desired bits from (or transforming the desired bits in) a variable by using logical bitwise operations. The operand (a constant or variable) that is used to perform masking is called a *mask*.

Masking is used in many different ways:

- To decide the bit pattern of an integer variable.
- To copy a portion of a given bit pattern to a new variable, while the remainder of the new variable is filled with 0s (using bitwise AND)
- To copy a portion of a given bit pattern to a new variable, while the remainder of the new variable is filled with 1s (using bitwise OR).
- To copy a portion of a given bit pattern to a new variable, while the remainder of the original bit pattern is inverted within the new variable (using bitwise exclusive OR).

The following function uses a mask to display the bit pattern of a variable:

```
#include <limits.h>
void bit_pattern(int u)
{
    int i, x, word;
    unsigned mask = 1;
```

```
    word = CHAR_BIT * sizeof(int);
    mask = mask << (word - 1);    /* shift 1 to the leftmost position */
    for(i = 1; i <= word; i++)
    {
        x = (u & mask) ? 1 : 0;  /* identify the bit */
        printf("%d", x);         /* print bit value */
        mask >>= 1;              /* shift mask to the right by 1 bit */
    }
}
```

## Section 4.4: Short circuit behavior of logical operators

Short circuiting is a functionality that skips evaluating parts of a (if/while/...) condition when able. In case of a logical operation on two operands, the first operand is evaluated (to true or false) and if there is a verdict (i.e first operand is false when using &&, first operand is true when using ||) the second operand is not evaluated.

Example:

```
#include <stdio.h>

int main(void) {
  int a = 20;
  int b = -5;

  /* here 'b == -5' is not evaluated,
     since a 'a != 20' is false. */
  if (a != 20 && b == -5) {
    printf("I won't be printed!\n");
  }

  return 0;
}
```

Check it out yourself:

```
#include <stdio.h>

int print(int i) {
  printf("print function %d\n", i);
  return i;
}

int main(void) {
  int a = 20;

  /* here 'print(a)' is not called,
     since a 'a != 20' is false. */
  if (a != 20 && print(a)) {
    printf("I won't be printed!\n");
  }

  /* here 'print(a)' is called,
     since a 'a == 20' is true. */
  if (a == 20 && print(a)) {
    printf("I will be printed!\n");
  }

  return 0;
}
```

Output:

```
$ ./a.out
print function 20
I will be printed!
```

Short circuiting is important, when you want to avoid evaluating terms that are (computationally) costly. Moreover, it can heavily affect the flow of your program like in this case: Why does this program print "forked!" 4 times?

# Section 4.5: Comma Operator

Evaluates its left operand, discards the resulting value, and then evaluates its rights operand and result yields the value of its rightmost operand.

```
int x = 42, y = 42;
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
```

The comma operator introduces a sequence point between its operands.

Note that the *comma* used in functions calls that separate arguments is NOT the *comma operator*, rather it's called a *separator* which is different from the *comma operator*. Hence, it doesn't have the properties of the *comma operator*.

The above `printf()` call contains both the *comma operator* and the *separator*.

```
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
/*           ^        ^ this is a comma operator */
/*           this is a separator */
```

The comma operator is often used in the initialization section as well as in the updating section of a `for` loop. For example:

```
for(k = 1; k < 10; printf("\%d\\n", k), k += 2);   /*outputs the odd numbers below 9/*

/* outputs sum to first 9 natural numbers */
for(sumk = 1, k = 1; k < 10; k++, sumk += k)
    printf("\%5d\%5d\\n", k, sumk);
```

# Section 4.6: Arithmetic Operators

**Basic Arithmetic**

Return a value that is the result of applying the left hand operand to the right hand operand, using the associated mathematical operation. Normal mathematical rules of commutation apply (i.e. addition and multiplication are commutative, subtraction, division and modulus are not).

**Addition Operator**

The addition operator (`+`) is used to add two operands together. Example:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;
```

```
    int c = a + b; /* c now holds the value 12 */

    printf("%d + %d = %d",a,b,c); /* will output "5 + 7 = 12" */

    return 0;
}
```

**Subtraction Operator**

The subtraction operator (`-`) is used to subtract the second operand from the first. Example:

```
#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 7;

    int c = a - b; /* c now holds the value 3 */

    printf("%d - %d = %d",a,b,c); /* will output "10 - 7 = 3" */

    return 0;
}
```

**Multiplication Operator**

The multiplication operator (`*`) is used to multiply both operands. Example:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a * b; /* c now holds the value 35 */

    printf("%d * %d = %d",a,b,c); /* will output "5 * 7 = 35" */

    return 0;
}
```

*Not to be confused with the `*` dereference operator.*

**Division Operator**

The division operator (`/`) divides the first operand by the second. If both operands of the division are integers, it will return an integer value and discard the remainder (use the modulo operator `%` for calculating and acquiring the remainder).

If one of the operands is a floating point value, the result is an approximation of the fraction.

Example:

```
#include <stdio.h>

int main (void)
```

```
{
    int a = 19 / 2 ; /* a holds value 9   */
    int b = 18 / 2 ; /* b holds value 9   */
    int c = 255 / 2; /* c holds value 127 */
    int d = 44 / 4 ; /* d holds value 11  */
    double e = 19 / 2.0 ; /* e holds value 9.5   */
    double f = 18.0 / 2 ; /* f holds value 9.0   */
    double g = 255 / 2.0; /* g holds value 127.5 */
    double h = 45.0 / 4 ; /* h holds value 11.25 */

    printf("19 / 2 = %d\n", a);    /* Will output "19 / 2 = 9"    */
    printf("18 / 2 = %d\n", b);    /* Will output "18 / 2 = 9"    */
    printf("255 / 2 = %d\n", c);   /* Will output "255 / 2 = 127" */
    printf("44 / 4 = %d\n", d);    /* Will output "44 / 4 = 11"   */
    printf("19 / 2.0 = %g\n", e);  /* Will output "19 / 2.0 = 9.5"    */
    printf("18.0 / 2 = %g\n", f);  /* Will output "18.0 / 2 = 9"      */
    printf("255 / 2.0 = %g\n", g); /* Will output "255 / 2.0 = 127.5" */
    printf("45.0 / 4 = %g\n", h);  /* Will output "45.0 / 4 = 11.25"  */

    return 0;
}
```

**Modulo Operator**

The modulo operator (`%`) receives integer operands only, and is used to calculate the remainder after the first operand is divided by the second. Example:

```
#include <stdio.h>

int main (void) {
    int a = 25 % 2;    /* a holds value 1  */
    int b = 24 % 2;    /* b holds value 0  */
    int c = 155 % 5;   /* c holds value 0  */
    int d = 49 % 25;   /* d holds value 24 */

    printf("25 % 2 = %d\n", a);     /* Will output "25 % 2 = 1"    */
    printf("24 % 2 = %d\n", b);     /* Will output "24 % 2 = 0"    */
    printf("155 % 5 = %d\n", c);    /* Will output "155 % 5 = 0"   */
    printf("49 % 25 = %d\n", d);    /* Will output "49 % 25 = 24"  */

    return 0;
}
```

**Increment / Decrement Operators**

The increment (`a++`) and decrement (

`a--`

) operators are different in that they change the value of the variable you apply them to without an assignment operator. You can use increment and decrement operators either before or after the variable. The placement of the operator changes the timing of the incrementation/decrementation of the value to before or after assigning it to the variable. Example:

```
#include <stdio.h>

int main(void)
{
    int a = 1;
    int b = 4;
```

```
    int c = 1;
    int d = 4;

    a++;
    printf("a = %d\n",a);    /* Will output "a = 2" */
    b--;
    printf("b = %d\n",b);    /* Will output "b = 3" */

    if (++c > 1) { /* c is incremented by 1 before being compared in the condition */
        printf("This will print\n");    /* This is printed */
    } else {
        printf("This will never print\n");    /* This is not printed */
    }

    if (d-- < 4) {  /* d is decremented after being compared */
        printf("This will never print\n");    /* This is not printed */
    } else {
        printf("This will print\n");    /* This is printed */
    }
}
```

As the example for `c` and `d` shows, both operators have two forms, as prefix notation and postfix notation. Both have the same effect in incrementing (`++`) or decrementing (`--`) the variable, but differ by the value they return: prefix operations do the operation first and then return the value, whereas postfix operations first determine the value that is to be returned, and then do the operation.

Because of this potentially counter-intuitive behaviour, the use of increment/decrement operators inside expressions is controversial.

# Section 4.7: Access Operators

The member access operators (dot `.` and arrow `->`) are used to access a member of a `struct`.

**Member of object**

Evaluates into the lvalue denoting the object that is a member of the accessed object.

```
struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
myObject.x = 42;
myObject.y = 123;

printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Outputs ".x = 42, .y = 123". */
```

**Member of pointed-to object**

Syntactic sugar for dereferencing followed by member access. Effectively, an expression of the form `x->y` is shorthand for `(*x).y` — but the arrow operator is much clearer, especially if the structure pointers are nested.

```
struct MyStruct
{
    int x;
    int y;
```

```
};

struct MyStruct myObject;
struct MyStruct *p = &myObject;

p->x = 42;
p->y = 123;

printf(".x = %i, .y = %i\n", p->x, p->y); /* Outputs ".x = 42, .y = 123". */
printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Also outputs ".x = 42, .y = 123". */
```

### Address-of

The unary `&` operator is the address of operator. It evaluates the given expression, where the resulting object must be an lvalue. Then, it evaluates into an object whose type is a pointer to the resulting object's type, and contains the address of the resulting object.

```
int x = 3;
int *p = &x;
printf("%p = %p\n", (void *)&x, (void *)p); /* Outputs "A = A", for some implementation-defined A.
*/
```

### Dereference

The unary `*` operator dereferences a pointer. It evaluates into the lvalue resulting from dereferencing the pointer that results from evaluating the given expression.

```
int x = 42;
int *p = &x;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 42, *p = 42". */

*p = 123;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 123, *p = 123". */
```

### Indexing

Indexing is syntactic sugar for pointer addition followed by dereferencing. Effectively, an expression of the form `a[i]` is equivalent to `*(a + i)` — but the explicit subscript notation is preferred.

```
int arr[] = { 1, 2, 3, 4, 5 };
printf("arr[2] = %i\n", arr[2]); /* Outputs "arr[2] = 3". */
```

### Interchangeability of indexing

Adding a pointer to an integer is a commutative operation (i.e. the order of the operands does not change the result) so `pointer + integer == integer + pointer`.

A consequence of this is that `arr[3]` and `3[arr]` are equivalent.

```
printf("3[arr] = %i\n", 3[arr]); /* Outputs "3[arr] = 4". */
```

Usage of an expression `3[arr]` instead of `arr[3]` is generally not recommended, as it affects code readability. It tends to be a popular in obfuscated programming contests.

# Section 4.8: sizeof Operator

**With a type as operand**

Evaluates into the size in bytes, of type `size_t`, of objects of the given type. Requires parentheses around the type.

```
printf("%zu\n", sizeof(int)); /* Valid, outputs the size of an int object, which is platform-dependent. */
printf("%zu\n", sizeof int); /* Invalid, types as arguments need to be surrounded by parentheses! */
```

**With an expression as operand**

Evaluates into the size in bytes, of type `size_t`, of objects of the type of the given expression. The expression itself is not evaluated. Parentheses are not required; however, because the given expression must be unary, it's considered best practice to always use them.

```
char ch = 'a';
printf("%zu\n", sizeof(ch)); /* Valid, will output the size of a char object, which is always 1 for all platforms. */
printf("%zu\n", sizeof ch);  /* Valid, will output the size of a char object, which is always 1 for all platforms. */
```

# Section 4.9: Cast Operator

Performs an *explicit* conversion into the given type from the value resulting from evaluating the given expression.

```
int x = 3;
int y = 4;
printf("%f\n", (double)x / y); /* Outputs "0.750000". */
```

Here the value of `x` is converted to a `double`, the division promotes the value of `y` to `double`, too, and the result of the division, a `double` is passed to `printf` for printing.

# Section 4.10: Function Call Operator

The first operand must be a function pointer (a function designator is also acceptable because it will be converted to a pointer to the function), identifying the function to call, and all other operands, if any, are collectively known as the function call's arguments. Evaluates into the return value resulting from calling the appropriate function with the respective arguments.

```
int myFunction(int x, int y)
{
    return x * 2 + y;
}

int (*fn)(int, int) = &myFunction;
int x = 42;
int y = 123;

printf("(*fn)(%i, %i) = %i\n", x, y, (*fn)(x, y)); /* Outputs "fn(42, 123) = 207". */
printf("fn(%i, %i) = %i\n", x, y, fn(x, y)); /* Another form: you don't need to dereference explicitly */
```

# Section 4.11: Increment / Decrement

The increment and decrement operators exist in *prefix* and *postfix* form.

```
int a = 1;
int b = 1;
int tmp = 0;

tmp = ++a;        /* increments a by one, and returns new value; a == 2, tmp == 2  */
tmp = a++;        /* increments a by one, but returns old value; a == 3, tmp == 2 */
tmp = --b;        /* decrements b by one, and returns new value; b == 0, tmp == 0 */
tmp = b--;        /* decrements b by one, but returns old value; b == -1, tmp == 0 */
```

Note that arithmetic operations do not introduce sequence points, so certain expressions with `++` or `--` operators may introduce undefined behaviour.

# Section 4.12: Assignment Operators

Assigns the value of the right-hand operand to the storage location named by the left-hand operand, and returns the value.

```
int x = 5;      /* Variable x holds the value 5. Returns 5. */
char y = 'c';   /* Variable y holds the value 99. Returns 99
                 * (as the character 'c' is represented in the ASCII table with 99).
                 */
float z = 1.5;  /* variable z holds the value 1.5. Returns 1.5. */
char const* s = "foo"; /* Variable s holds the address of the first character of the string 'foo'.
*/
```

Several arithmetical operations have a *compound assignment* operator.

```
a += b  /* equal to: a = a + b */
a -= b  /* equal to: a = a - b */
a *= b  /* equal to: a = a * b */
a /= b  /* equal to: a = a / b */
a %= b  /* equal to: a = a % b */
a &= b  /* equal to: a = a & b */
a |= b  /* equal to: a = a | b */
a ^= b  /* equal to: a = a ^ b */
a <<= b /* equal to: a = a << b */
a >>= b /* equal to: a = a >> b */
```

One important feature of these compound assignments is that the expression on the left hand side (`a`) is only evaluated once. E.g if `p` is a pointer

```
*p += 27;
```

dereferences `p` only once, whereas the following does so twice.

```
*p = *p + 27;
```

It should also be noted that the result of an assignment such as `a = b` is what is known as an *rvalue*. Thus, the assignment actually has a value which can then be assigned to another variable. This allows the chaining of assignments to set multiple variables in a single statement.

This *rvalue* can be used in the controlling expressions of `if` statements (or loops or `switch` statements) that guard

some code on the result of another expression or function call. For example:

```
char *buffer;
if ((buffer = malloc(1024)) != NULL)
{
    /* do something with buffer */
    free(buffer);
}
else
{
    /* report allocation failure */
}
```

Because of this, care must be taken to avoid a common typo which can lead to mysterious bugs.

```
int a = 2;
/* ... */
if (a = 1)
    /* Delete all files on my hard drive */
```

This will have disastrous results, as `a = 1` will always evaluate to `1` and thus the controlling expression of the `if` statement will always be true (read more about this common pitfall here). The author almost certainly meant to use the equality operator (`==`) as shown below:

```
int a = 2;
/* ... */
if (a == 1)
    /* Delete all files on my hard drive */
```

**Operator Associativity**

```
int a, b = 1, c = 2;
a = b = c;
```

This assigns `c` to `b`, which returns `b`, which is than assigned to `a`. This happens because all assignment-operators have right associativity, that means the rightmost operation in the expression is evaluated first, and proceeds from right to left.

# Section 4.13: Logical Operators

**Logical AND**

Performs a logical boolean AND-ing of the two operands returning 1 if both of the operands are non-zero. The logical AND operator is of type `int`.

```
0 && 0  /* Returns 0. */
0 && 1  /* Returns 0. */
2 && 0  /* Returns 0. */
2 && 3  /* Returns 1. */
```

**Logical OR**

Performs a logical boolean OR-ing of the two operands returning 1 if any of the operands are non-zero. The logical OR operator is of type `int`.

```
0 || 0  /* Returns 0. */
```

```
0 || 1  /* Returns 1.  */
2 || 0  /* Returns 1.  */
2 || 3  /* Returns 1.  */
```

**Logical NOT**

Performs a logical negation. The logical NOT operator is of type `int`. The NOT operator checks if at least one bit is equal to 1, if so it returns 0. Else it returns 1;

```
!1 /* Returns 0. */
!5 /* Returns 0. */
!0 /* Returns 1. */
```

**Short-Circuit Evaluation**

There are some crucial properties common to both `&&` and `||`:

- the left-hand operand (LHS) is fully evaluated before the right-hand operand (RHS) is evaluated at all,
- there is a sequence point between the evaluation of the left-hand operand and the right-hand operand,
- and, most importantly, the right-hand operand is not evaluated at all if the result of the left-hand operand determines the overall result.

This means that:

- if the LHS evaluates to 'true' (non-zero), the RHS of `||` will not be evaluated (because the result of 'true OR anything' is 'true'),
- if the LHS evaluates to 'false' (zero), the RHS of `&&` will not be evaluated (because the result of 'false AND anything' is 'false').

This is important as it permits you to write code such as:

```
const char *name_for_value(int value)
{
    static const char *names[] = { "zero", "one", "two", "three", };
    enum { NUM_NAMES = sizeof(names) / sizeof(names[0]) };
    return (value >= 0 && value < NUM_NAMES) ? names[value] : "infinity";
}
```

If a negative value is passed to the function, the `value >= 0` term evaluates to false and the `value < NUM_NAMES` term is not evaluated.

# Section 4.14: Pointer Arithmetic

**Pointer addition**

Given a pointer and a scalar type `N`, evaluates into a pointer to the `N`th element of the pointed-to type that directly succeeds the pointed-to object in memory.

```
int arr[] = {1, 2, 3, 4, 5};
printf("*(arr + 3) = %i\n", *(arr + 3)); /* Outputs "4", arr's fourth element. */
```

It does not matter if the pointer is used as the operand value or the scalar value. This means that things such as `3 + arr` are valid. If `arr[k]` is the `k+1` member of an array, then `arr+k` is a pointer to `arr[k]`. In other words, `arr` or `arr+0` is a pointer to `arr[0]`, `arr+1` is a pointer to `arr[2]`, and so on. In general, `*(arr+k)` is same as `arr[k]`.

Unlike the usual arithmetic, addition of 1 to a pointer to an `int` will add 4 bytes to the current address value. As array names are constant pointers, `+` is the only operator we can use to access the members of an array via pointer notation using the array name. However, by defining a pointer to an array, we can get more flexibility to process the data in an array. For example, we can print the members of an array as follows:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", *(arr + k));
    }
    return 0;
}
```

By defining a pointer to the array, the above program is equivalent to the following:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr; /* or int *ptr = &arr[0]; */
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", ptr[k]);
        /* or   printf("\n\t%d", *(ptr + k)); */
        /* or   printf("\n\t%d", *ptr++); */
    }
    return 0;
}
```

See that the members of the array `arr` are accessed using the operators `+` and `++`. The other operators that can be used with the pointer `ptr` are `-` and `--`.

**Pointer subtraction**

Given two pointers to the same type, evaluates into an object of type `ptrdiff_t` that holds the scalar value that must be added to the second pointer in order to obtain the value of the first pointer.

```
int arr[] = {1, 2, 3, 4, 5};
int *p = &arr[2];
int *q = &arr[3];
ptrdiff_t diff = q - p;

printf("q - p = %ti\n", diff); /* Outputs "1". */
printf("*(p + (q - p)) = %d\n", *(p + diff)); /* Outputs "4". */
```

# Section 4.15: _Alignof

Version ≥ C11

Queries the alignment requirement for the specified type. The alignment requirement is a positive integral power of 2 representing the number of bytes between which two objects of the type may be allocated. In C, the alignment requirement is measured in `size_t`.

The type name may not be an incomplete type nor a function type. If an array is used as the type, the type of the array element is used.

This operator is often accessed through the convenience macro `alignof` from **`<stdalign.h>`**.

```
int main(void)
{
    printf("Alignment of char = %zu\n", alignof(char));
    printf("Alignment of max_align_t = %zu\n", alignof(max_align_t));
    printf("alignof(float[10]) = %zu\n", alignof(float[10]));
    printf("alignof(struct{char c; int n;}) = %zu\n",
            alignof(struct {char c; int n;}));
}
```

Possible Output:

```
Alignment of char = 1
Alignment of max_align_t = 16
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

http://en.cppreference.com/w/c/language/_Alignof