## Variables

Declaring variable

```
int num;
char flag_a, flag_b;
```

Initializing variable

```
int num = 12;
char flag_a = 'a', flag_b = 'b';
```

Assigning variable

```
num = 12;
flag_a = 'a', flag_b = 'b';
```

Variable name must follow these rules:
- Contains only alphabets, numbers and underscore (_)
- Cannot start with numbers
- Cannot be a reserved keyword. Eg: `if`, `for`, `int`

Variables are case sensitive
`var_name` ≠ `Var_Name`

## Data Types

| Group | Type | Value range |
|---|---|---|
| Character types | `char` | -128 to 127 |
| Signed integer types | `short` | -32,768 to 32,767 |
| | `int` | -2,147,483,648<br>to<br>2,147,483,647 |
| | `long` | |
| | `long long` | -9,223,372,036,854,775,808<br>to<br>9,223,372,036,854,775,807 |
| Unsigned integer types | `unsigned short` | 0 to 65,535 |
| | `unsigned int` | 0 to 4,294,967,295 |
| | `unsigned long` | |
| | `unsigned long long` | 0<br>to 18,446,744,073,709,551,615 |
| Floating-point types | `float` | |
| | `double` | |
| | `long double` | |
| Boolean type | `bool` | true or false |
| Void type | `void` | |

## Operators

Assignment operator

| | |
|---|---|
| = | Assign a value to a variable |

Arithmetic operator

| | |
|---|---|
| + | Add two values |
| - | Subtract two values |
| * | Multiply two values |
| / | Divide two values |
| % | Get remainder of divison |

Compound assignment

| | |
|---|---|
| += | Add and assign to variable |
| -= | Subtract and assign to variable |
| *= | Multiply and assign to variable |
| /= | Divide and assign to variable |
| %= | Get remainder of divison<br>and assign to variable |

Increment and decrement

| | |
|---|---|
| ++ | Increase value by 1 |
| -- | Decrease value by 1 |

```
x = 3;
y = ++x;

// x = 4, y = 4
```

```
x = 3;
y = x++;

// x = 4, y = 3
```

Logical operator

| | |
|---|---|
| ! | Invert condition (NOT) |
| && | Requires both<br>condition to be true (AND) |
| \|\| | Requires either<br>condition to be true (OR) |

Relational and comparison operator

| | |
|---|---|
| == | Equal to |
| != | Not equal to |
| < | Less than |
| > | More than |
| <= | Less than or equal |
| >= | More than or equal |

Conditional ternary operator

```
condition ? value_if_true : value_if_false
```

## Flow control

```
if (x) {
    // do something...
}
```

If x is true, performs all statement inside this block.

```
else if (y) {
    // do something else...
}
```

(Optional) Otherwise, check if y is true.
If so, perform all statement in this block.

```
else {
    // if all else fails, do this...
}
```

(Optional) If none of the above statements ran, perform all statement in this block instead.

Parts of the if-statement may be left out if not needed.

The following works:

```
if (x) {...}

if (x) {...}
else {...}

if (x) {...}
else if (y){...}
```

```
while (x) {
    // do something
}
```

As long as x is true,
perform all statement in this block.

```
do {
    // do something
} while (x);
```

Same as regular `while` loop,
but runs the block at least once
even if condition is not met.

Once per iteration, checks if condition is true.

The loop will execute **all** statement inside the scope, even if condition is false halfway.

```
int x = 3;
while (x > 0) {
    x = -1;
    // Below statement still runs
    cout << "Hello";
}
```

```
for (initialization; condition; expression) {
    // do something
}

for (int x = 0; x < 10; x++) {
    // 0 1 2 3 4 5 6 7 8 9
    cout << x << ' ';
}
```

Designed to loop a **known** number of times.

Note that the initialization, condition and expression are optional.
Omiting all 3 will result in an endless loop.

```
for ( ; ; ) {
    // Unless there is a break or return,
    // this will loop infinitely.
}
```

The condition is checked once per iteration, and
the expression is performed at the end of each iteration.

## `while` vs `for`

### while

The number of times
to loop cannot be found easily

```
int num;
cout << "Enter number: ";
cin >> num;

while (num != 0) {
    cout << "Enter another: ";
    cin >> num;
}
```

We don't know how many inputs
the user will provide before entering 0.

### for

The number of times
to loop can be found easily

```
string msg = "Good day";
for (int i = 0; i < msg.length(); i++)
{
    cout << msg[i];
}
```

Number of characters in string
can be easily found using `.length()`

## References

References is an **alias** for an existing variable.
Any operation done on the reference is
directly applied to the original variable.

```
int a = 5;
int& ref = a; // 'ref' is a reference to 'a'

ref += 3; // Modifies 'a' through the reference
cout << a; // Prints 8
```

A reference variable must be initialized when declared

```
int& my_ref; // This will give an error
```

Once initialized, it cannot be change to refer
to another variable

```
int a = 5;
int b = 10;
int& ref = a; // 'ref' refers to 'a'

// Does NOT make 'ref' reference 'b',
// it simply sets 'a' to 10
ref = b;
```

## Pointers

Pointers stores the **memory address** of another variable.
It holds the address where a value is stored in memory.

```
int a = 5;
int* ptr = &a; // 'ptr' stores the address of 'a'

// Dereference pointer using * to
// access value at the memory address
*ptr += 3;
cout << *ptr; // Prints 8
```

To change the memory address of a pointer,
assign the address of the new variable

```
int a = 5;
int b = 10;
int* ptr = &a;

// Note that the dereference
// operator is NOT here.
// We want change address, not value.
ptr = &b;
cout << *ptr; // Prints 10
```

## Pass-by-Value, Pass-by-Reference, Pass-by-Pointer

In **pass-by-value**, a copy of the argument is passed to the function.
Changes to the argument will not affect the original variable.

```
void modify(int x) {
    x = 10; // Modifies the copy only
}

int main() {
    int a = 5;
    modify(a);
    // 'a' remains 5 because only a copy was modified
}
```

In **pass-by-reference**, the argument is a reference to the original variable.
Changes to the argument will affect the original variable.

```
void modify(int& x) {
    x = 10; // Modifies the original variable
}

int main() {
    int a = 5;
    modify(a);
    // 'a' is now 10 because it was modified
    // through the reference

    // Keep in mind the function argument
    // must be a reference to a variable.
    // The below will result in error.
    modify(5);
}
```

In **pass-by-pointer**, a **copy** of the memory address of the variable is passed
to the function. Dereference it to access or modify the original variable.

```
void modify(int* x) {
    *x = 10; // Dereference to modify the original variable
}

int main() {
    int a = 5;
    modify(&a); // Pass the address of 'a'.
    // 'a' is now 10 because it was
    // modified through the pointer
}
```

Because its a copy of the memory address, changing the address
in the function will not change the address of the original variable.

```
void modify(int* x) {
    int b = 1;
    x = &b; // This doesn't change the original variable

    // 'x' now points to address of 'b', but the original
    // variable still points to address of 'a'
}

int main() {
    int a = 5;
    modify(&a); // 'a' is still 5.
}
```

## Array

A series of elements in contiguous memory locations.

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

Indexes in array starts at 0

Initializing an array

```
int x[5] = {10, 20, 30, 40, 50};
```

Number of elements (size)

Size of array is fixed at declaration

```
int x[5] = {10, 20, 30, 40, 50, 60};
```

Excess elements will cause error

Access an element of an array using subscript operator ([])

```
int num = x[3];
```

Index to access

Be careful not to access out-of-bound indexes.
It results in undefined behavior.

```
int x[5] = {10, 20, 30, 40, 50};
int num = x[5];
```

No such index.
May or may not cause error.

Array names represents a constant pointer to the first element,
so you cannot assign an array to another array.

```
int x[5] = {10, 20, 30, 40, 50};
int y[5] = {0};

// This will cause an error
y = x;
```

You must copy each element of the old array
into the new array instead.

For functions, array arguments will decay into pointers

```
// All three functions are identical
void my_function(int my_array[])
void my_function(int my_array[10])
void my_function(int *my_array)
```

As a function argument, the size
will be lost. I.e: size doesn't matter.

## Vector #include <vector>

Like array, but with some differences:

Can dynamically grow or shrink

```
vector<int> vec; // Empty vector that can grow as needed
vec.push_back(10); // Automatically resizes
```

Has bounds checking with `.at()`

```
int arr[5] = {0};
arr[10] = 20; // Undefined behavior (out of bounds)

vector<int> vec(5);
vec.at(10) = 20; // Causes error
vec[10] = 20; // This one is still undefined behavior
```

Can lexicographically compare the content of two vectors.
First, compare by their content. If both matches, then compare their size.

```
vector<int> vec_a = {0, 1, 2};
vector<int> vec_b = {0, 1, 2, 3};

if (vec_a <= vec_b) {
    // This block will run.
}
```

```
int arr_a[3] = {0, 1, 2};
int arr_b[3] = {0, 1, 2};
// This block will never run
if (arr_a == arr_b) {
    // Array comparison always
    // evaluates to false
}
```

### Element access

Access specified element

```
reference at(int pos)
reference operator[](int pos)
```

Access the first element

```
reference front()
```

Access the last element

```
reference back()
```

### Capacity

Checks if vector is empty

```
bool empty()
```

Returns the number of elements

```
int size()
```

### Modifiers

Clears the vector

```
void clear()
```

Removes the last element

```
void pop_back()
```

Adds an element to the end

```
void push_back(type value)
```

Swap content and capacity

```
void swap(vector& other)
```

## Scope

Scope determines the visibility and lifetime of variables, functions, etc.
Nested scope can access all of their outer scopes, but not the other way around.

```
int x = 5; // 'x' is visible to all nested scope
x = y - 1; // 'y' is NOT visible here, this will cause an error

void my_func() {
    int y = 2; // 'y' is visible to only the scope of this function
    y += x; // 'x' is visible here, so this works
}
```

In this example,
the variable 'x' is declared in the outer scope, and
the variable 'y' is declared in the nested function scope.

'x' is visible inside of `my_func`. But,
'y' is not visible outside of `my_func`.

### Variable Shadowing

This occurs when a variable in both the nested scope and the outer scope shares the **same variable name**.

The variable inside the nested scope will "shadow" or "hide" the variable in the outer scope.

```
int x = 5; // Outer scope variable

void my_func() {
   int x = 10; // Nested scope variable

   // Note that although both variable have the
   // same name, the nested scope variable will
   // be used inside my_func.
   cout << x; // Prints 10.
}
```

### Local scope

Variables defined within a block have **local scope**.
Those variables are only accessible within the block.

Those variables are destroyed when the block ends.

```
void my_func() {
    int x = 5; // 'x' has local scope within my_func
    // 'x' is only accessible inside my_func
}
// 'x' is not accessible here, outside my_func
```

### Function scope

Variables declared within a function's parameter list have **function scope**.
Those variables are only accessible within the function body and are destroyed once the function returns.

```
void my_func(int y) { // 'y' has function scope
    // 'y' is only accessible inside my_func
}
// 'y' is not accessible here, outside my_func
```

### Global scope

Variables declared outside of all functions have **global scope**.
Global variables can be access from any part of the entire .cpp file.

They exists for the duration of the program and are destroyed when the program ends.

```
int z = 10; // 'z' has global scope

void my_func() {
    z = 20; // 'z' can be accessed and modified inside any function
}
```

## Characters

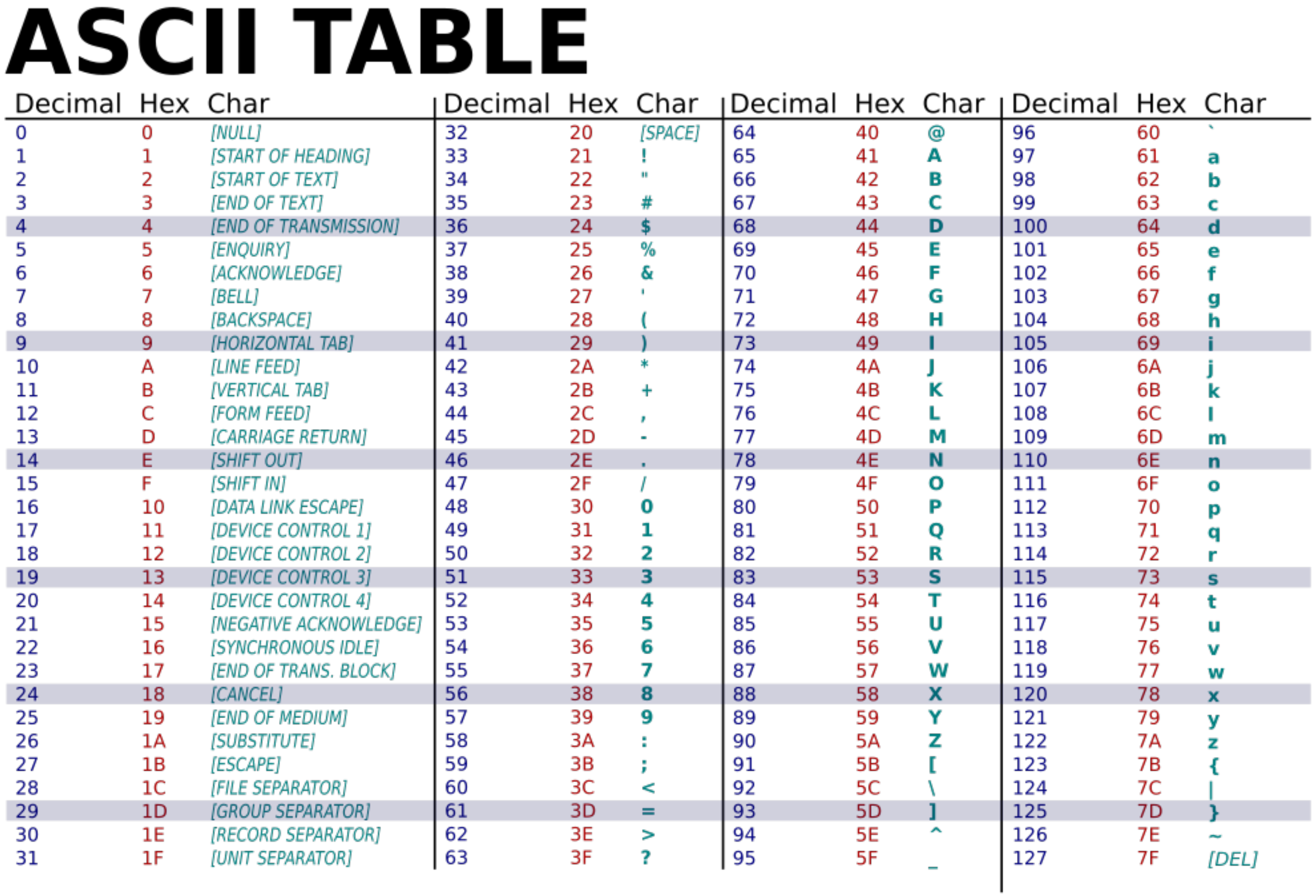
Each character literal is mapped to a value in the ASCII table. (See below)

Because of this, it is possible to apply arithmetic operations on characters as if they were numbers.

```
char c = 'a'; // According to ASCII table, letter 'a' is 97.
c += 3; // Variable is now 97+3 (100), which is letter 'd'.
```

For most part, it is possible to treat a character **as an integer**.
This includes loops.

```
for (char c = 'A'; c <= 'Z'; ++c) {
    cout << c; // Prints out all the alphabets in ascending order.
}
```



# ASCII TABLE

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [END OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

## #include <cmath>

### Basic operations

Returns absolute value (|x|)

```
int abs(int num)
long abs(long num)
long long abs(long long num)
float abs(float num)
double abs(double num)
long double abs(long double num)
```

Returns the larger of two floating point values

```
float fmax(float x, float y)
double fmax(double x, double y)
long double fmax(long double x, long double y)
```

Returns the smaller of two floating point values

```
float fmin(float x, float y)
double fmin(double x, double y)
long double fmin(long double x, long double y)
```

Returns the positive difference between x and y

```
float fdim(float x, float y)
double fdim(double x, double y)
long double fdim(long double x, long double y)
```

Returns remainder of x/y
(x - iquot * y, where iquot is x/y rounded towards zero)

```
float fmod(float x, float y)
double fmod(double x, double y)
long double fmod(long double x, long double y)
```

Returns remainder of x/y
(x - quo * y, where quo is x/y rounded half to even)

```
float remainder(float x, float y)
double remainder(double x, double y)
long double remainder(long double x, long double y)
```

### Rounding operations

Round to nearest integer, away from zero in halfway cases

```
float round(float num)
double round(double num)
long double round(long double num)
```

Round up to nearest integer
(negative values will remove decimal part)

```
float ceil(float num)
double ceil(double num)
long double ceil(long double num)
```

Round down to nearest integer
(negative values rounds to next largest negative integer)

```
float floor(float num)
double floor(double num)
long double floor(long double num)
```

Remove decimal part

```
float trunc(float num)
double trunc(double num)
long double trunc(long double num)
```

### Exponential operations

Raises a number to the given power ($x^y$)

```
float pow(float x, float y)
double pow(double x, double y)
long double pow(long double x, long double y)
```

Returns cube root of x

```
float cbrt(float x)
double cbrt(double x)
long double cbrt(long double x)
```

Returns common logarithm of x ($\log_{10}$ x)

```
float log10(float x)
double log10(double x)
long double log10(long double x)
```

Returns square root of x

```
float sqrt(float x)
double sqrt(double x)
long double sqrt(long double x)
```

Returns natural logarithm of x (ln x)

```
float log(float x)
double log(double x)
long double log(long double x)
```

## #include <string>

```
string greeting = "Hello";
```

### Concatenation

Join strings using + operator

```
string firstName = "John";
string lastName = "Doe";

string fullName = firstName + " " + lastName;
```

Beware: Adding two strings is not the same as adding two numbers

```
string x = "10", y = "20";
string z = x + y; // z will be 1020 (as string)
```

### String Length

Get length of a string using length()

```
string txt = "ABCDEFGHIJ";
txt.length(); // returns 10
```

### Access String as char

Access characters inside a string using []

```
string msg = "Message";
for (int i = 0; i < msg.length(); i++)
{
    cout << msg[0];
}
```

### Escaping Characters

Print illegal characters using \

```
string book = "Read \"The Book\" now";
```

### Comparing 2 strings

Compare each character from left to right

```
string str1 = "apple";
string str2 = "apricot";

if (str1 < str2) {
    // Output will display this.
    cout << "str1 comes before str2.\n";
} else {
    cout << "str1 comes after str2.\n";
}
```

# printf() vs cout

## printf()

### Formatting

Offers wide range of formatting options

```
// Output: Width 5:     7

int y = 7;
printf("Width 5: %5d\n", y);
```

Print with a minimum of 5 characters

## cout

### Type safety

Automatically determines the data types

```
char tada = 'p';
cout << tada << '\n';
```

No need for specifiers (%c),
cout applies the appropriate formatting

## #include <cstdio>

Returns the number of characters written
(on error, returns a negative value)

```
int printf(const char* format, ...)
```

| | |
|---|---|
| %% | Writes literal % |
| %c | Writes a single character |
| %s | Writes a character string |
| %d or %i | Converts a signed integer into decimal representation |
| %o | Converts an unsigned integer into octal representation |
| %x or %X | Converts an unsigned integer into hexadecimal representation |
| %u | Converts an unsigned integer into decimal representation |
| %f or %F | Converts floating-point number into decimal representation |
| %e or %E | Converts floating-point number to the decimal exponent notation |
| %a or %A | Converts floating-point number to the hexadecimal exponent notation |

## #include <iostream>

Display output to standard output device
(in a console app, defaults to the screen)

```
extern std::ostream cout
```

Accept input from standard input device
(in a console app, defaults to the keyboard)

```
extern std::istream cin
```

```
int num;

cout << "Enter a number: ";

// take integer input
cin >> num;

cout << "You entered: " << num << '\n';
cout << "Exiting...";
```