Collections: vector

```
int main() {
    vector<int> v; // v can only hold integers
    cout << "v.size(): " << v.size() << endl;

    v.push_back(17);
    v.push_back(42);
    cout << "v.size(): " << v.size() << endl;

    for (size_t i = 0; i < v.size(); ++i) {
        cout << v[i] << ' ';
    }
    cout << endl;

    v.clear();
    cout << "v.size(): " << v.size() << endl;
}
```

- **#include <vector>**
- *Similar to* Python's list and Java's ArrayList
- *"Generic"* type
- Other handy methods: back(), pop_back(), capacity()

How can we print in reverse?

vector initialization

```
int main() {

    // Initialize the size of the vector and the value of that all the
    // entries will start with.
    // The vector v1 will be of size 7, with every entry equal to 42
    vector<int> v1(7, 42); // parentheses

    // specify the distinct values to initialize each entry to
    vector<int> v2{1, 1, 2, 3, 5, 8, 13, 21};  // curly braces

}
```

ranged for (1)

```
int main() {
    vector<int> v;

            …

    // for (size_t i = 0; i < v.size(); ++i) {
    //     cout << v[i] << ' ';
    // }

    for (int value : v) {
        cout << value << ' ';
    }
    cout << endl;

            …
}
```

- Don’t need the index?
- Then easier to use a “ranged for”
- Similar to Python’s for
- Requires C++11 and higher

ranged for (2)

```
int main() {
    vector<int> v{2, 3, 5, 7, 11};

    for (int value : v) {
        value = 42;
    }

    for (int value : v) {
        cout << value << ' ';
    }
    cout << endl;
}
```

- What gets printed?
- Not 42’s…
- We will see how to fix this next time

Prefix and postfix

- `++` and `--` operators can be used in complex statements and expressions
- In prefix mode (`++val, --val`) the operator increments or decrements, ***then*** returns the value of the variable
- In postfix mode (`val++, val--`) the operator returns the value of the variable, ***then*** increments or decrements

Prefix and postfix - examples

```
int num, val = 12;
cout << val++;

cout << ++val;

num = --val;

num = val--;
```

Prefix and postfix - examples

```
int num, val = 12;
cout << val++;          // displays 12,
                        // val is now 13;
cout << ++val;          // sets val to 14,
                        // then displays it
num = --val;            // sets val to 13,
                        // stores 13 in num
num = val--;            // stores 13 in num,
                        // sets val to 12
```

Prefix and postfix – cont.

- Can be used in expressions:

```
result = num1++ + --num2;
```

- Must be applied to something that has a location in memory. Cannot have:

```
result = (num1 + num2)++;
```

- Can be used in relational expressions:

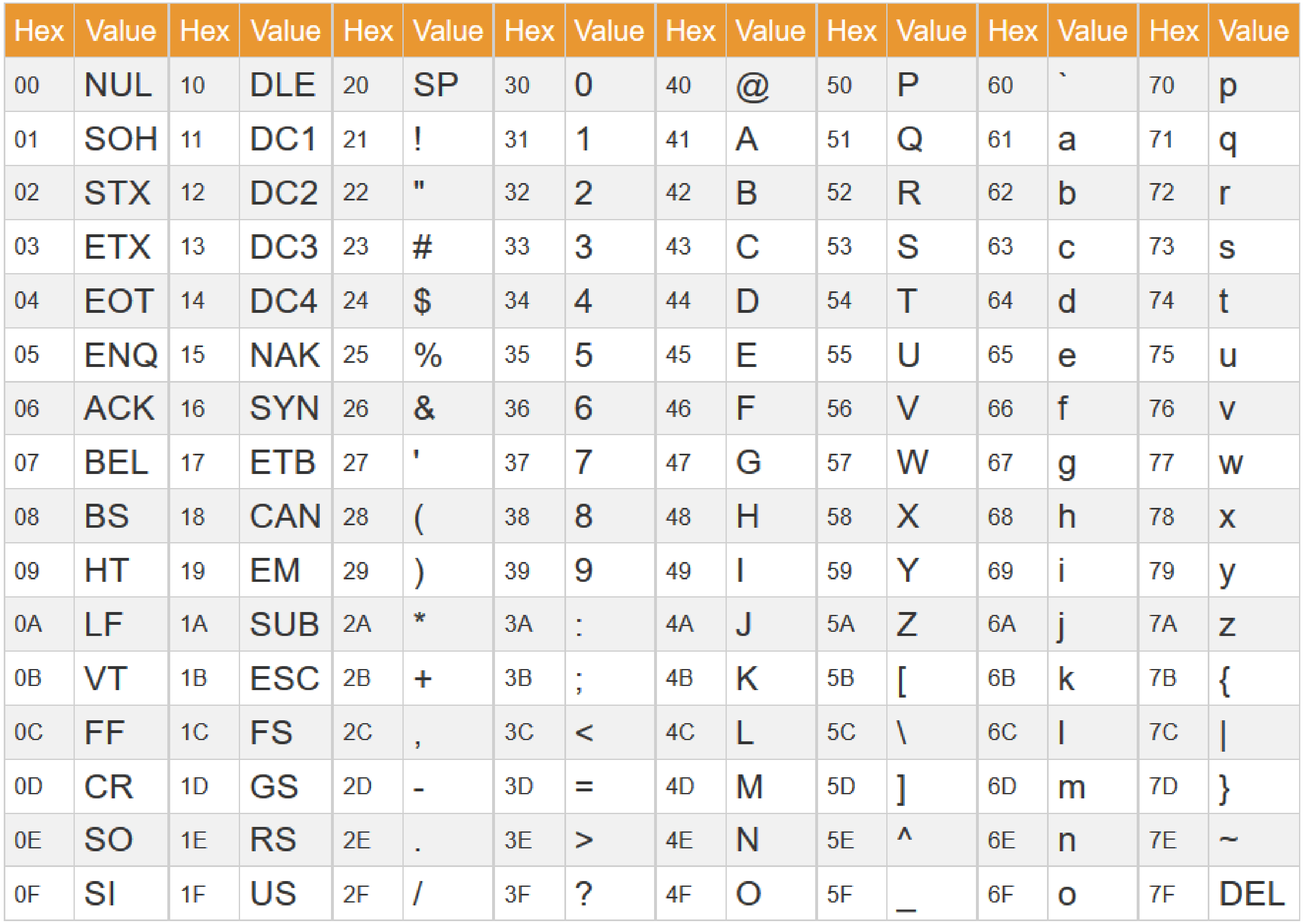
```
if (++num > limit)
```

pre- and post-operations will cause different comparisons

Char - primitive data type

A character literal uses *single* quotes.

```
char c = 'q';
```



Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Chars - example

What is the value stored?
‘a’, a, 61 or 97

```
#include <iostream>
#include <string>
using namespace std;

int main() {

    char c = ‘a’;
    cout << c << endl;

    c = ‘a’ + 1;
    cout << c << endl;

    char c2 = 78; // Happens to be ascii for 'N'
    cout << c2 << endl;

    int n = 79;
    c2 = n;
    cout << c2 << endl;

    n = 100000;
    c2 = n;
    cout << c2 << endl;
    //    c2 = 334;
    //    cout << c2 << endl;

}
```

```
a
b
N
O
á
```

- A character variable may be assigned an integer value within the range that will fit a char.
  - The size of char is 1 byte, thus it can hold values from -128 to 127.
- If we attempt to assign an integer value that the compiler can see is too large, then we will get a warning.

strings

- A **non-primitive** type → thus use
#include <string>

- String **literals** use *double* quotes
  - string s = "this is a string"

- A string variable acts very much like a vector of characters
  - push_back, pop_back, size, clear

- with some additional useful features.
  - E.g. can input or output a string, but not a vector

string example(1)

```
int main() {
    string s1 = “Hello world, ";
    cout << s1 << endl;

    for (char c : s1) {
        cout << c << ' ';
    }
    cout << endl;

    string s2 = “this is CS2124";

    string s3 = s1 + s2;
    cout << s3 << endl;
}
```

```
Hello world,
H e l l o   w o r l d ,
Hello world, this is CS2124
```

- Strings can be looped over the same as vectors
- The plus operator concatenates strings

strings are *mutable*!

```
int main() {
    string animal = "bat";

    // What is the type of animal[0]?
    animal[0] += 1;

    // What will be output?
    cout << animal << endl;
}
```

```
cat
```

- The elements of a `string` are of type `char`.
- `string`s are *mutable*. We can modify their contents

File I/O

- **#include <fstream>**
- Opening and closing
- Testing for successful open
- Reading input
- Looping over a file

Open, test for success, read, close

```
int main() {
    ifstream jab("jabberwocky");
    if (!jab) {
        cerr << "failed to open jabberwocky";
        exit(1);
    }

    string something;
    jab >> something;
    cout << something << endl;

    jab.close();
}
```

```
Twas brillig and the slithey toves
did gyre and gymble in the wabe.
All mimsy were the borogroves
and the momeraths outgrabe.

Beware the Jabberwock, my son!
The jaws that bite, the claws that
catch!
Beware the JubJub bird and shun
The frumious Bandersnatch!


...
```

```
Twas
```

Open, test for success, read a line, close

```
int main() {
    ifstream jab("jabberwocky");
    if (!jab) {
        cerr << "failed to open jabberwocky";
        exit(1);
    }

    string something;
    getline(jab, something);
    cout << something << endl;

    jab.close();
}
```

```
Twas brillig and the slithey toves
did gyre and gymble in the wabe.
All mimsy were the borogroves
and the momeraths outgrabe.

Beware the Jabberwock, my son!
The jaws that bite, the claws that
catch!
Beware the JubJub bird and shun
The frumious Bandersnatch!


...
```

```
Twas brillig and the slithey toves
```

Read token by token

```
int main() {
    ifstream jab("jabberwocky");
    if (!jab) {
        cerr << "failed to open jabberwocky";
        exit(1);
    }

    string something;
    while (jab >> something) {
        cout << something << endl;
    }

    jab.close();
}
```

- We put the read into the while condition
- Magically the loop stops when there is nothing more to read
- Each read gets the next ***whitespace delimited*** token.

Vector of lines

```
int main() {
    ifstream jab("jabberwocky");
    string line;
    vector<string> lines;

    while (getline(jab, line)) {
        lines.push_back(line);
    }
    jab.close();

    // print the contents of the vector
    for (size_t i = 0; i < lines.size(); ++i) {
        cout << lines[i] << endl;
    }

    // print them in reverse?
    for (size_t i = lines.size(); i > 0 ; --i) {
        cout << lines[i-1] << endl;
    }
}
```

- Omitting test of open to save space
- Close the file as soon as we're done with it
- The while loop continues so long as getline is successful

Vector of strings as 2D data structure

```
int main() {
    ifstream jab("jabberwocky");
    string line;
    vector<string> lines;

    while (getline(jab, line)) {
        lines.push_back(line);
    }
    jab.close();

    for (size_t i = 0; i < lines.size(); ++i) {
        for (size_t j = 0; j < lines[i].size(); ++j) {
            cout << lines[i][j] << ' ';
        }
        cout << endl;
    }
}
```

```
Twas brillig and the slithey toves
did gyre and gymble in the wabe.
All mimsy were the borogroves
and the momeraths outgrabe.

Beware the Jabberwock, my son!
The jaws that bite, the claws that
catch!
Beware the JubJub bird and shun
The frumious Bandersnatch!


…
```

Declaring a variable as `const`

- The contents of variables declared with the const keywords cannot be changed
  ex
  ```
  const double TAX_RATE = 0.0675;
  const int NUM_STATES = 50;
  ```

- variables that are constant ? Sounds like a contradiction?
  Linguistically yes. But a variable is nothing but a memory location, and hence it is possible to assign it a value once, and declare that it cannot be modified.

- `const` variables must be initialized during declaration/definition.

Vector of vectors as 2-dimensional data structure

```
int main() {
    const int ROWS = 10;
    const int COLS = 10;
    vector<vector<int>> mat;

    for (int r = 0; r < ROWS; ++r) {
        mat.push_back(vector<int>(COLS));
        for (int c = 0; c < COLS; ++c) {
            mat[r][c] = r * COLS + c;
        }
    }

    for (int r = 0; r < ROWS; ++r) {
        for (int c = 0; c < COLS; ++c) {
            cout << mat[r][c] << ' ';
        }
        cout << endl;
    }
}
```

- Filling 2d "matrix" with values from 0 to 99
- **Constants** are **commonly** in all caps.
- Note creation of temporary vector being pushed onto mat.

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

Variable scope

- The scope of a variable:

  The part of the program in which the variable can be accessed (**i.e. its visibility**)

- A variable cannot be used before it is defined
- Variables may be local, global, static local or static global.

Variable scope – cont.

- A ***local variable*** is **visible only in the code block** in which it is defined, from the point of definition to the end of the block.

- A code block may be defined by two curly braces { }, e.g.
  - Code within a function (e.g. main()),
  - An if statement, case statement, a for/while/do-while loop,
  - Any code block defined by two curly braces without any preceding if, else or case statements, or any function definition.

- ***Global variables*** are variables declared outside functions. Hence they are not inside any code block, and thus have global visibility.

Variable scope – cont.

```
#include <iostream>
using namespace std;

int main() {
    {
        int x = 20;
    }
    cout << x << endl;

}
```

What is the “reference” type?

- A reference is a variable that refers to another variable. It is declared using the & sign, e.g.

```
int x=12;                 // an integer variable
int &rx = x;              // rx is a reference variable
```

- In this example, **`rx` is a reference to `x` and any value read/written to `rx` is also read/written to `x`**.

- It is widely common for compilers to implement references as pointers that are implicitly dereferenced in each use. The C++ Standard does not force compilers to implement references using pointers however.
    - **Pointers will be studied later**

- References must be ***initialized during declaration*** and may not be modified thereafter **EXCEPT** when used in a ranged-for loop.

Ranged for – by value

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
        vector<int> myvec{ 0,1,2,3,4 };

        // increment each element by 1
        for (int val : myvec) {
                val++;
        }

        // print the entire vector
        for (int val : myvec) {
                cout << val << ' ‘;
        }
        cout << endl;

        return 0;
}
```

```
0 1 2 3 4
```

Ranged for – by ref

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
        vector<int> myvec{ 0,1,2,3,4 };

        // increment each element by 1
        for (int &val : myvec) {
                val++;
        }

        // print the entire vector
        for (int val : myvec) {
                cout << val << ' ‘;
        }
        cout << endl;

        return 0;
}
```

```
1 2 3 4 5
```

Ranged for – by const ref

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
	vector<int> myvec{ 0,1,2,3,4 };

	// increment each element by 1
	for (const int &val : myvec) {
		val++;
	}

	// print the entire vector
	for (int val : myvec) {
		cout << val << ' ‘;
	}
	cout << endl;

	return 0;
}
```

Compilation error

Function prototypes

- Ways to notify the compiler about a function (must take place before using the function in your code)

1) Place function definition before using it.

2) Use a function prototype (i.e., a function declaration, does not contain the function's body)

  - Prototype:

```
void printHeading();
```

  - Definition:

```
void printHeading(){
    …
    …
}
```

Notice the difference?

Function prototyping

Function prototype example:

```
void myfunc(int, int, double);
```

arguments

Function call example:

```
myfunc(5,6,7.1);
```

parameters

Function definition example:

```
void myfunc(int x, int y, int z){
    cout << x << ", " << y << ", " << z << endl;
}
```

Pass-by-value versus pass-by-reference

- When calling a function, the arguments may be passed by value or by reference.

- <u>In pass-by-value semantics</u>, when an argument is passed to a function, its value is copied into the parameter.
  - Changes to the parameter in the function ***do not affect*** the value of the argument.

- <u>In pass-by-reference semantics</u>, an argument is passed to a function as a reference.
  - Changes to a parameter in the function ***directly change*** the value of the caller’s argument.

Pass-by-ref

```
#include <iostream>

using namespace std;

int incr(int& x);

int main()
{
    int y = 5;
    cout << "Func: " << incr(y) << endl;
    cout << "Value of y: " << y << endl;
    return 0;
}

int incr(int& x) {
    return ++x;
}
```

```
Func: 6
Value of y: 6
```

int is converted into int&

- Functions must be either defined or prototyped prior to usage.
- Argument passed to reference parameter must be a variable – cannot be an expression or constant
- Use only when function body intends to modify the arguments.

pass by value

```
#include <iostream>

using namespace std;

int incr(int x);

int main()
{
    int y = 5;
    cout << "Func: " << incr(y) << endl;
    cout << "Value of y: " << y << endl;
    return 0;
}

int incr(int x) {
    return ++x;
}
```

```
Func: 6
Value of y: 5
```

When function body does not intend to modify the arguments:

- Use a constant reference (Then if you mistakenly insert a code that modifies the argument → compilation error!

OR

- Pass by value

pass by const ref

```
#include <iostream>

using namespace std;

int incr(const int& x);

int main()
{
    int y = 5;
    cout << "Func: " << incr(y) << endl;
    cout << "Value of y: " << y << endl;
    return 0;
}

int incr(const int& x) {
    return ++x;
}
```

Compilation error

When function body does not intend to modify the arguments:

- Use a constant reference (Then if you mistakenly insert a code that modifies the argument → compilation error!

OR

- Pass by value