CheatSheet by Florian Frauenfelder, Computer Science Math/Phys AS 2023

0 Misc

0.1 Number conversion

Numbers of different types in joint expressions are converted to the most general or "biggest" one according to the following order:

```
bool / char < int < unsigned int < float < double
```

0.2 Operator precedence

Operator	Prec.	Dir.
`::`	17	L
`a++, f(), [], ., ->`	16	L
`++a, -a, !, ~, (T), *p, &a, new, delete`	15	R
`*, /, %`	13	L
`a+b, a-b`	12	L
`<, <=, >, >=`	9	L
`==, !=`	8	L
`&, ^, |`	7, 6, 5	L
`&&`	4	L
`||`	3	L
`=, +=, -=, *=, /=, %=, &=, ^=, |=`	2	R

0.3 Hex, decimal and binary

Hex	Bin	Dec	Hex	Bin	Dec
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	a	1010	10
3	0011	3	b	1011	11
4	0100	4	c	1100	12
5	0101	5	d	1101	13
6	0110	6	e	1110	14
7	0111	7	f	1111	15

Powers of 2:

0	1	2	3	4	5	6	7	8	9	10
1	2	4	8	16	32	64	128	256	512	1024

Multiples of 16:

3	4	5	6	7	8	9	10	11	12	13	14	15	16
48	64	80	96	112	128	144	160	176	192	208	224	240	256

0.4 ASCII

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
32	20		48	30	0	64	40	@	80	50	P	96	60	‘	112	70	p
33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
39	27	’	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

The first characters (0 – 31 or 00 – 1F) are non-printable control characters.

1 Integers

1.1 Signed and negative integers

Negative integers can be displayed in binary using the following method: Write down all possible binary combinations in order. Half of them start with 1, we use this first bit as the **sign**. Now interpret all numbers with the first bit set as negative numbers, with the highest one $(11\ldots11)$ corresponding to -1. This way, we get $2^{\text{bits}-1}-1$ positive and $2^{\text{bits}-1}$ negative numbers, with the remaining one being $0 \cong 00\ldots00$.

Example with three bits:

Dec	Bin	Bin	Dec
0	000	000	0
1	001	111	-1
2	010	110	-2
3	011	101	-3
		100	-4

2 Logical values

2.1 Short-circuit evaluation

C++ uses so-called **short-circuit evaluation**:

- `a && b`: If `a` evaluates to `false`, `b` is not evaluated.
- `a || b`: If `a` evaluates to `true`, `b` is not evaluated.

3 Floating point numbers

3.1 Floating point system

We describe a **floating point system**

$$F(\beta, p, e_{\min}, e_{\max})$$

with four integers, $\beta \geq 2$ being the **base**, $p \geq 1$ the **significand**, $e_{\min}$ the smallest and $e_{\max} \geq e_{\min}$ the largest possible **exponent**. This system contains the numbers

$$\pm d_0.d_1 d_2 \dots d_{p-1} \cdot \beta^e \in F(\beta, p, e_{\min}, e_{\max})$$

where $d_0 \neq 0$ and $d_i \in \{0, \dots, \beta - 1\}$ and $e \in \{e_{\min}, \dots, e_{\max}\}$.

`C++` uses $F(2, 24, -126, 127)$ for `float` and $F(2, 53, -1022, 1023)$ for `double`. This means, for `float`, which is `32-bit`, we have 1 bit for the sign, 23 bits for the significand (because the first bit is always one, we do not need to specify this) and 8 bit for the exponent, which means 256 different values, of which only 254 $(127 - (-126) + 1)$ are used, and the two remaining are reserved for special values like $0, \infty, \dots$ and the like.

With these systems, we need to be careful, because many numbers do not have exact representations in `C++`:

```
1.1 - 1.0 != 0.1
1.1 - 1.0 != 1.1f - 1.0f
```

But, all integers can be converted to floating point numbers without loss of precision.

From this also result the following three important rules:

1. Don't test rounded floating point numbers for equality.
2. Don't add floating point numbers with big differences in the exponent.
3. Don't subtract floating point numbers of similar exponents.

4 Recursion

4.1 Important concepts

A recursive function *always* needs a **base case**, and when the function is called, the base case must be reached at some point. Otherwise, we get infinite recursion and stack-overflow (→ exit code `-11` on Code Expert).

There are two main recursive strategies: '**Decrease and conquer**' and '**Divide and conquer**'. We can get to the base case if we always decrease by one and calculate the problem with $n - 1$, or we can get to the base case if we divide and calculate two remaining, smaller problems with similar sizes, that do not depend on each other. Dividing often is more memory- and time-efficient, but more complicated to write and understand. (With dividing also comes the possibility of parallelizing.)

Recursive solutions often lead to more elegant, simpler and shorter solutions, while iterative solutions are more efficient.

4.2 Examples from lecture slides

4.2.1 Factorial

Decrease and conquer:

```
unsigned int fac(unsigned int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n-1);
}
```

4.2.2 Euclidean Algorithm

Decrease and conquer?

```
unsigned int gcd(unsigned int a, unsigned int b) {
  if (b == 0)
    return a;
  else
    return gcd(b, a % b);
}
```

4.2.3 Bitstrings

Decrease and conquer:

```
void bs(std::string& bs, unsigned int i) {
  if (i == bs.size()) {
    std::cout << bs << "\n";
  } else {
    bs[i] = '0';
    all_bs(bs, i+1);
    bs[i] = '1';
    all_bs(bs, i+1);
  }
}
```

4.2.4 Vector-sum

Decrease and conquer:

```
int sum(const std::vector<int>& v, unsigned int from) {
  if (from >= v.size())
    return 0;
  else
    return v.at(from) + sum(v, from + 1);
}
```

Or Divide and conquer:

```cpp
using uint = unsigned int;
int sum(const std::vector<int>& v, uint from, uint to) {
  if (from > to)
    return 0;
  else if (from == to)
    return v.at(from);
  else {
    uint middle = (from + to) / 2;
    return sum(v, from, middle) + sum(v, middle + 1, to);
  }
}
```

5 Function overloading

Functions in C++ can be **overloaded** by defining a new function with the *same name*, but a *different* ***signature***, meaning a different number and/or order of arguments and/or arguments with different types (the names of the arguments do not influence the signature). The correct implementation of the function when called is chosen using the provided values for the arguments.

Important: Defining two functions with the same signature results in a **compiler error**.

6 Custom data types

6.1 Operator overloading

With most operators, everything is straightforward. But:

```cpp
T& operator+=(T& a, T b) {
  // calculations
  return a;
}
```

and similarly for `-=`, `*=`, `/=`, `%=`.

And:

```cpp
// POST: r has been written to out
std::ostream& operator<<(std::ostream& out, T t) {
  return out << /* something with t */;
}
```

and:

```cpp
// PRE:  in starts with something for t
// POST: t has been read from in
std::istream& operator>>(std::istream& in, T& t) {
  return in >> /* something with t */;
}
```

With out- and input, best practice is to use similar to- and from-string-conversion formats for consistency.

6.2 Structs and Classes

The only difference between a `struct` and a `class` is that in a `struct`, everything is `public` by default, and in a `class`, everything is `private` by default.

6.3 Containers

The standard library of C++ contains many useful **container** implementations. Some examples are:

- `array<T, size>`, an ordered unsorted collection implemented as a static array (→ contiguous memory locations)
- `vector<T>`, an ordered unsorted collection implemented as a dynamic array (→ contiguous memory locations)
- `list<T>`, an ordered unsorted collection implemented as a doubly linked list
- `unordered_set<T>`, an unordered duplicate-free collection implemented as a hash table
- `set<T>`, an ordered unsorted duplicate-free collection implemented as a red-black tree

6.4 Iterators

Every container in C++ has its **iterator** with the following properties:

- `it = c.begin()` is an iterator pointing to the first element.
- `it = c.end()` is an iterator pointing *behind* the last element.
- `++it` moves the iterator to the next element.
- `*it` de-references the iterator, returning the value of the element (see pointers).

Therefore, we can implement many helpful functions and methods with containers, without knowing their exact implementation, just with iterators.

7 Dynamic Memory and pointers

"**Pointers** are like **references** that may point to nothing".

7.1 Operators and types

- `T* p` means a pointer to data of type `T`.
- `&a` returns a pointer to `a`.
- `*p` de-references the pointer `p` and returns the value of the data behind the pointer.

7.2 const-(mad)ness

Rule: Read from right to left.

`int const p1`	`p1` is a constant integer
`int const* p2`	`p2` is a pointer to a constant integer
`int* const p3`	`p3` is a constant pointer to an integer
`int const* const p4`	`p4` is a constant pointer to a constant integer

We see that the "only" important thing is the placement of the `*`, everything else can actually be swapped around.

Remark: `const` with methods in classes is different: `T C::f() const = {}` means that the method `f` does not change any of the values of the instance of `C`.

7.3 new and delete

Rule: For every `new`, there must be a corresponding `delete`.

`T* p = new T()` returns a *pointer* to the created instance of `T`. The object lives until explicitly deleted with `delete p`.

But: `delete p` does not set `p = nullptr`!

Always be careful with `new` and `delete`: dangling pointers, zombie-memory, memory leaks, etc ...

To simplify these problems, we can use `std::shared_ptr<T>` or `std::unique_ptr<T>`.

7.4 Rule of three

Rule: If a class implements one of

- destructor
- copy constructor
- assignment operator,

it also needs the other two!

7.4.1 Destructor

The **destructor** `C::~C()` contains the necessary corresponding `delete` statements for the `new` statements in the constructors.

7.4.2 Copy constructor

The **copy constructor** `C::C(const C& x)` constructs a "real" or **deep copy** of all member variables.

7.4.3 Assignment operator

The **assignment operator** `C& C::operator=(const C& c)` is very similar to the copy constructor, but first needs to deconstruct `this` and then copy from the other object.

Important: Make sure to not do anything in case of self-assignment.

The assignment operator can be implemented for example in the following two ways:

- Call destructor, (adapted) code from copy constructor, `return *this`
- Use temporary object with copy constructor from the other object, use `std::swap()` with all values on `this` and the temporary object, `return *this`