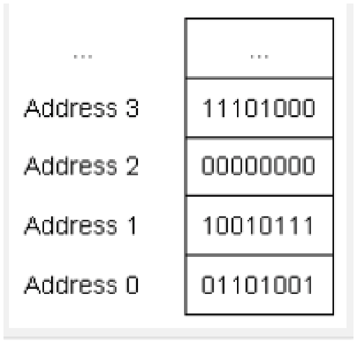
4 - Fundamental Data Types

4.1 - Introduction to Fundamental Data Types

Bits, Bytes and Memory Adressing

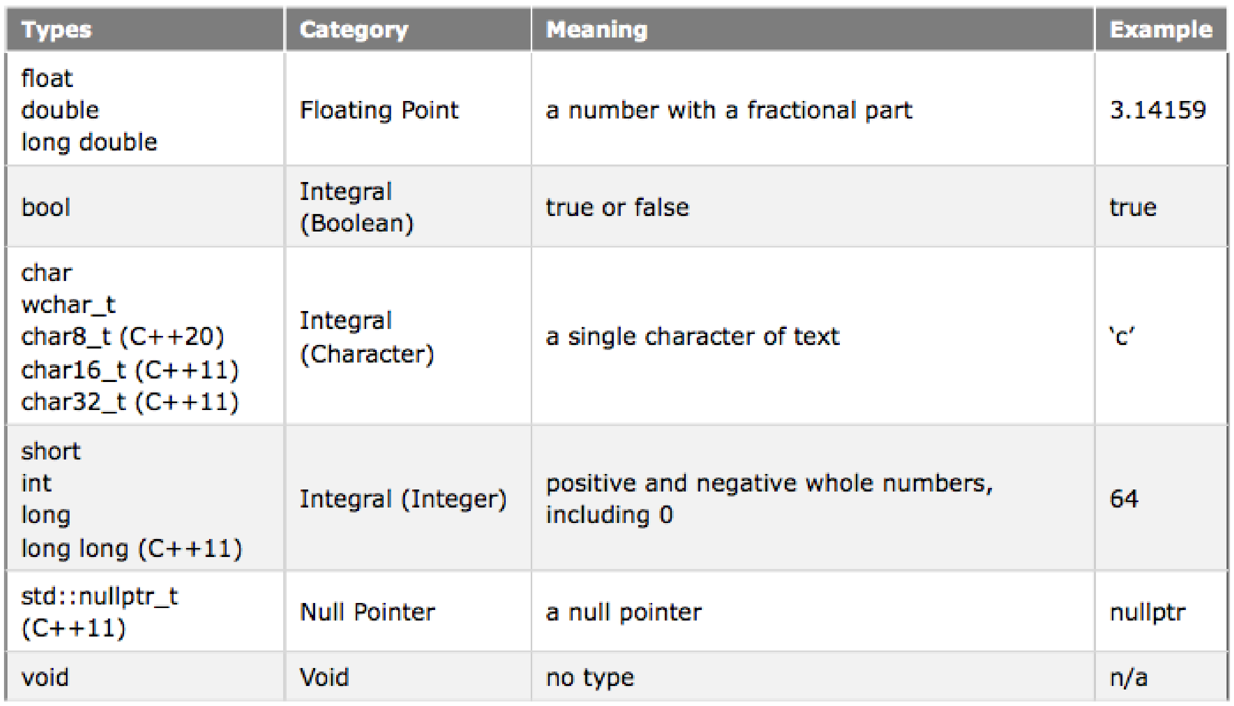
- Computers have RAM that's available for programs to use. When a variable is defined a piece of that RAM is set aside for that variable.
- The smallest unit of memory is a **binary digit** (called a **bit**), which can hold the value 0 or 1. Were you to take a random segment of memory, you'd see 011010100101010.
- Memory is organized into sequential units called memory address. allowing one to locate contents across the memory.



- In modern computer architectures each bit doesn't get its own memory - need to access data bit by bit is rare. Instead each memory address holds 1 byte - 8 sequential bits - of data.
  The picture shows a few sequential memory address
- Since all data on a computer is just a sequence of bits, we use data type (called "type" for short) to tell the compiler how to interpret contents of the memory in a meaningful way.
- If you assign an integer object the value 65, that value is converted to the sequence of bits 0100 0001. When the object is evaluated to produce a value, the sequence of bits is reconstituted back.

Fundamental data types

C++ comes with built-in support for many different data types, called fundamental data types, informally called basic types, primitive types, or built-in types.



Types	Category	Meaning	Example
float double long double	Floating Point	a number with a fractional part	3.14159
bool	Integral (Boolean)	true or false	true
char wchar_t char8_t (C++20) char16_t (C++11) char32_t (C++11)	Integral (Character)	a single character of text	'c'
short int long long long (C++11)	Integral (Integer)	positive and negative whole numbers, including 0	64
std::nullptr_t (C++11)	Null Pointer	a null pointer	nullptr
void	Void	no type	n/a

4.2 - Void

Void means "no-type", consequentially, variable cannot be defined with a type of void - they are used in other contexts.

- Void is often used with functions that don't return a value.

```
void writeValue(int x) // void here means no return value
{
    std::cout << "The value of x is: " << x << '\n';
    // no return statement, because this function doesn't return a value
}
```

- The use of keyword void is considered depreciated in C++. Therefore it's best to use an empty parameter list instead of void to indicate that a function doesn't take any parameters

```
int getValue() // empty function parameters is an implicit void
{
    int x;
    std::cin >> x;
    return x;
}
```

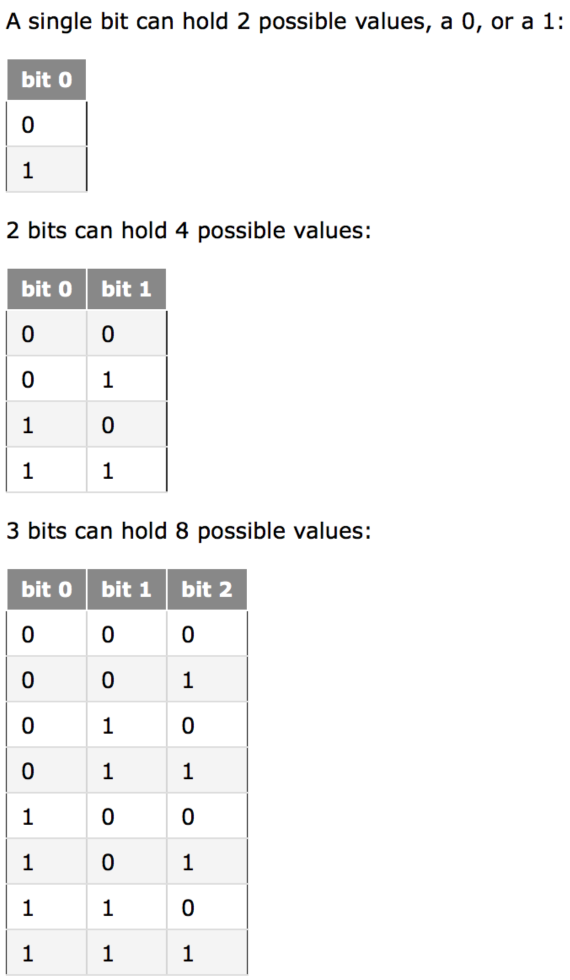
4.3 - Object Sizes and the Sizeof Operator

Object sizes

Memory on modern machines are organized into byte-sized units, with each byte of the memory having a unique address. And in most cases objects usually take up more than 1 byte memory - a single object may take up-to 2, 4, 8 or even more consecutive address. And usually information concerning the space taken by variables are hidden by the compiler,

- To generalize, an object with n bits can hold up 2^n different values.
- 2 bytes can hold up to 2^16 values.
- Every time we define an object a small portion fo that free memory is used for as long as the object is in existence..

> Focus on writing maintainable code and optimize only when and where the benefit of doing so is substantiated.



A single bit can hold 2 possible values, a 0, or a 1:

bit 0
0
1

2 bits can hold 4 possible values:

bit 0	bit 1
0	0
0	1
1	0
1	1

3 bits can hold 8 possible values:

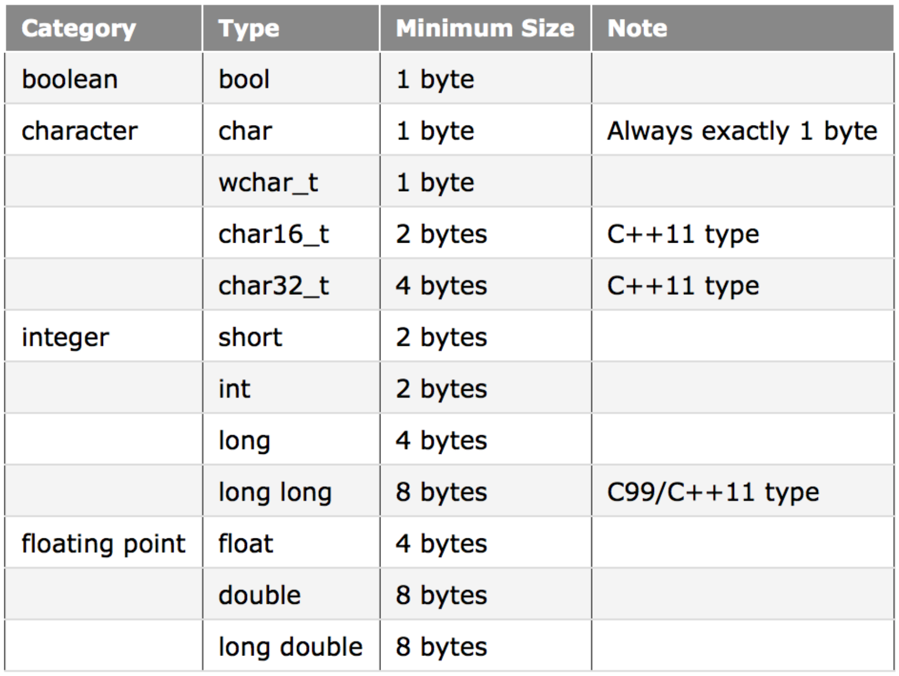
bit 0	bit 1	bit 2
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Fundamental data type sizes

- int short is usually a signed 32-bit objct (-32k to 32k). 2 bytes = 2^16 bits,
- int int is similar to short, but its

minimum size need not stay fixed at 2 bytes

- Float maintains a 32-bit precision, while it's 64 for double.



Category	Type	Minimum Size	Note
boolean	bool	1 byte	
character	char	1 byte	Always exactly 1 byte
	wchar_t	1 byte	
	char16_t	2 bytes	C++11 type
	char32_t	4 bytes	C++11 type
integer	short	2 bytes	
	int	2 bytes	
	long	4 bytes	
	long long	8 bytes	C99/C++11 type
floating point	float	4 bytes	
	double	8 bytes	
	long double	8 bytes	

Sizeof operator

- An unary operator that takes either a type or a variable and returns its size in bytes.

```
#include <iostream>
int main()
{
    std::cout << "bool:\t\t" << sizeof(bool) << " bytes\n";
    std::cout << "char:\t\t" << sizeof(char) << " bytes\n";
    std::cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes\n";
    std::cout << "char16_t:\t" << sizeof(char16_t) << " bytes\n'
    std::cout << "char32_t:\t" << sizeof(char32_t) << " bytes\n'
    std::cout << "short:\t\t" << sizeof(short) << " bytes\n";
    std::cout << "int:\t\t" << sizeof(int) << " bytes\n";
    std::cout << "long:\t\t" << sizeof(long) << " bytes\n";
    std::cout << "long long:\t" << sizeof(long long) << " bytes\
    std::cout << "float:\t\t" << sizeof(float) << " bytes\n";
    std::cout << "double:\t\t" << sizeof(double) << " bytes\n";
    std::cout << "long double:\t" << sizeof(long double) << " by

    return 0;
}
```

Output

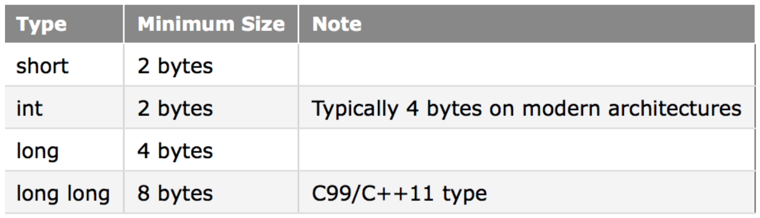
```
bool:           1 bytes
char:           1 bytes
wchar_t:        2 bytes
char16_t:       2 bytes
char32_t:       4 bytes
short:          2 bytes
int:            4 bytes
long:           4 bytes
long long:      8 bytes
float:          4 bytes
double:         8 bytes
long double:    8 bytes
```

On modern machines, objects of the fundamental data types are fast, so performance while using these types should generally not be a concern.

> You might assume that types that use less memory would be faster than types that use more memory. This is not always true. CPUs are often optimized to process data of a certain size (e.g. 32 bits), and types that match that size may be processed quicker. On such a machine, a 32-bit int could be faster than a 16-bit short or an 8-bit char.

4.4 - Signed Integers

An integer is a integral type that can represent +ve and -ve whole numbers, including 0. C++ has 4 different fundamental integer types as see in the image on the right.



Type	Minimum Size	Note
short	2 bytes	
int	2 bytes	Typically 4 bytes on modern architectures
long	4 bytes	
long long	8 bytes	C99/C++11 type

- By default, integers are **signed**. which means the number's sign is preserved, hence capable of holding +ve, and-ve numbers (and 0).
- Preferred way of defining 4 types of signed integers

```
short s;
int i;
long l;
long long ll;
```

- Integer types can also also an optional signed keyword, placed before the type name.

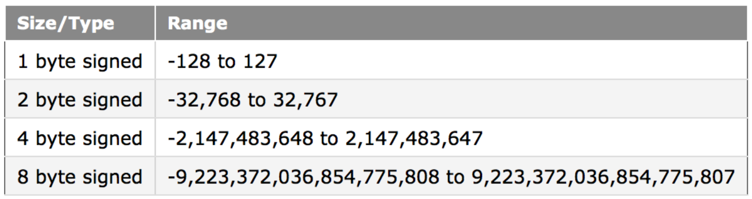
  ```
  signed short ss;
  signed int si;
  signed long sl;
  signed long long sll;
  ```

  T**he keywords should not be used, as it is redundant, since integers are signed by default**.

> Prefer the shorthand types that do not use the int suffix or signed prefix

Signed integer ranges

By definition, a 1-byte signed integer has a range of -128 to 127. This means a signed integer can store any integer value between -128 and 127. Were it unsigned, it would be between 0 and 256



Size/Type	Range
1 byte signed	-128 to 127
2 byte signed	-32,768 to 32,767
4 byte signed	-2,147,483,648 to 2,147,483,647
8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

- Signed range is defined by -(2^(n-1)) to 2^(n-1) - 1.

Integer overflow

Exactly what happens when we try to assign 289 to a 1-byte signed integer, which is outside the range of what a 1-byte signed integer can hold. Hence integer overflow is a phenomenon that occurs when we try to store a value that's outside the range of the type. **Signed integer overflow will result in undefined behavior**.

```
int main()
{
    signed short x{0}; // smallest 2-byte unsigned value possible
    std::cout << "x was: " << x << '\n';

    x = 38000; // -1 is out of our range, so we get wrap-around
    std::cout << "x is now: " << x << '\n';

    x = 32767; // -2 is out of our range, so we get wrap-around
    std::cout << "x is now: " << x << '\n';

    return 0;
}
// x was: 0
// x is now: -27536    (38000-32767) if > 0 : take quotient and substract -32,768 + 1 (zero)
// x is now: 32767
```

4.3 Unsigned Integers and Why to Avoid them

Unsigned integers are those that can only hold non-negative numbers

```
unsigned short us;
unsigned int ui;
unsigned long ul;
unsigned long long ull;  // An n-bit unsigned variable has a range of 0 to (2^n)-1.
```

Unsigned integer overflow

Say we were trying to store 280 (requiring 9-bits of storage) in 1-byte unsigned integer. One would say that this could result in overflow, But this doesn't happen, instead it is divided by one greater than the largest number of the type, and only the remainder kept.

```
int main()
{
    unsigned short x{ 65535 }; // largest 16-bit unsigned value possible
    std::cout << "x was: " << x << '\n';

    x = 65536; // 65536 is out of our range, so we get wrap-around
    std::cout << "x is now: " << x << '\n';

    x = 65537; // 65537 is out of our range, so we get wrap-around
    std::cout << "x is now: " << x << '\n';

    return 0;
}
// x was: 65535
// x is now: 0 // divided by 65535 + 1
// x is now: 1 // divided by 65535 + 1

int main()
{
  unsigned int x{ 3 };
  unsigned int y{ 5 };

  std::cout << x - y << '\n';
  return 0;
}
// 4294967294 -2 wrapping around to a number close to the top of the range of a 4-byte integer
// Additionaly, if one of the variables were signed, during the operation, both
// are turned unsigned to result in a similar behaviour
```

> Bjarne Stroustrup, the designer of C++, said, “Using an unsigned instead of an int to gain one more bit to represent positive integers is almost never a good idea

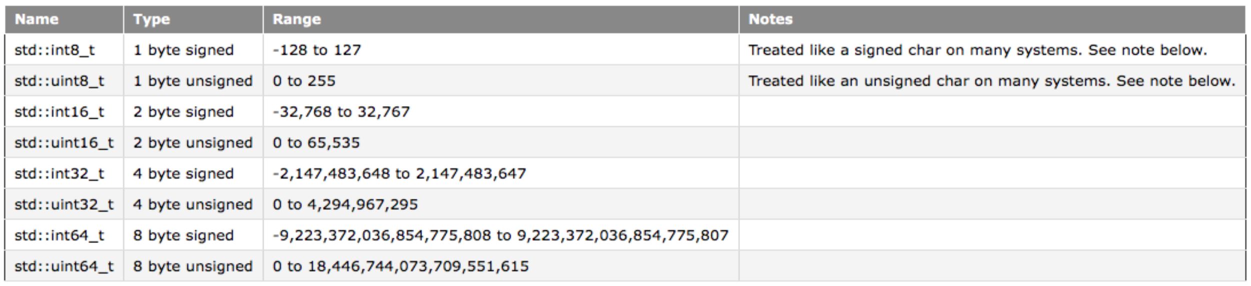
Where to use unsigned numbers?

- Preferred when dealing with bit-manipulation
- Performing array indexing
- Developing embedded systems

4.6 - Fixed-width Integers and Size_t

Fixed-width integers

To help with cross-platform portability, C99 defined a set of fixed-width integers (in the stdint.h header) that are guaranteed to have the same size on any architecture.



Name	Type	Range	Notes
std::int8_t	1 byte signed	-128 to 127	Treated like a signed char on many systems. See note below.
std::uint8_t	1 byte unsigned	0 to 255	Treated like an unsigned char on many systems. See note below.
std::int16_t	2 byte signed	-32,768 to 32,767	
std::uint16_t	2 byte unsigned	0 to 65,535	
std::int32_t	4 byte signed	-2,147,483,648 to 2,147,483,647	
std::uint32_t	4 byte unsigned	0 to 4,294,967,295	
std::int64_t	8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	
std::uint64_t	8 byte unsigned	0 to 18,446,744,073,709,551,615	

```
#include <iostream>
#include <cstdint>

int main()
{
    std::int16_t i(5); // direct initialization
    std::cout << i;
    return 0;
}
```

This being said, fixed-width integers have two **downsides**:

- They may **not be supported** on architectures where the defined types can't be represented.
- They may also be **less performant** than the built-in type in some architectures.

Fast and least integers

Fast and least integers help deal with the downsides of fixed-width integers.

- The fast type (std::int_fast#_t) provides the fastest signed integer, with a width of at least # bits (where # = 8, 16, 32, or 64). For example, std::int_fast32_t will give you the fastest signed integer type that's at least 32 bits.
- The least type (std::int_least#_t) provides the smallest signed integer type with a width of at least # bits (where # = 8, 16, 32, or 64). For example, std::int_least32_t will give you the smallest signed integer type that's at least 32 bits.

There is also an unsigned set of fast and least types (std::uint_fast#_t and std::uint_least#_t)

```
#include <iostream>
#include <cstdint>
```

```
int main()
{
  std::cout << "fast 8: " << sizeof(std::int_fast8_t) * 8 << " bits\n";
  std::cout << "fast 16: " << sizeof(std::int_fast16_t) * 8 << " bits\n";
  std::cout << "fast 32: " << sizeof(std::int_fast32_t) * 8 << " bits\n";

  std::cout << "least 8: " << sizeof(std::int_least8_t) * 8 << " bits\n";
  std::cout << "least 16: " << sizeof(std::int_least16_t) * 8 << " bits\n";
  std::cout << "least 32: " << sizeof(std::int_least32_t) * 8 << " bits\n";

  return 0;
}
// fast 8: 8 bits
// fast 16: 32 bits (fast 16 gives the fastest signed int to be 32 bits)
// fast 32: 32 bits
// least 8: 8 bits
// least 16: 16 bits (least 16 gives smallest signed int to be 16 bits)
// least 32: 32 bits
```

> Favor the std::int_fast#_t and std::int_least#_t integers when you need an integer guaranteed to be at least a certain minimum size.

std::int8_t and std::uint8_t may behave like chars

Due to an oversight in C++ specifications, most compilers define and treat std::int8_t and std::uint8_t (and the corresponding fast and least fixed-width types) identically to types signed char and unsigned char respectively.

```
int main()
{
    std::int8_t myint = 65;
    std::cout << myint;

    return 0;
}
// A On most systems, this program will print ‘A’
```

Therefore, avoid the 8-bit fixed-width integer types. If you do use them, note that they are often treated like chars.

4.8 - Floating point numbers

Scientific Notation

```
Start with: 42030
Slide decimal left 4 spaces: 4.2030e4
No leading zeros to trim: 4.2030e4
```

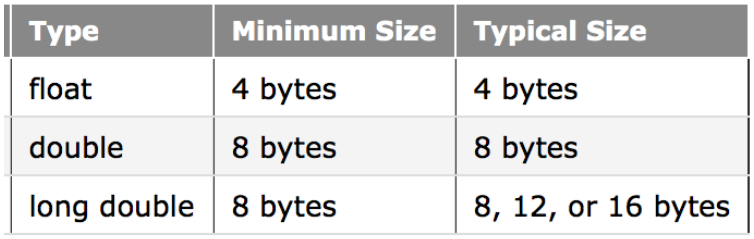
```
Trim trailing zeros: 4.203e4 (4 significant digits)

Start with: 0.0078900
Slide decimal right 3 spaces: 0007.8900e-3
Trim leading zeros: 7.8900e-3
Don't trim trailing zeros: 7.8900e-3 (5 significant digits)

Start with: 600.410
Slide decimal left 2 spaces: 6.00410e2
No leading zeros to trim: 6.00410e2
Don't trim trailing zeros: 6.00410e2 (6 significant digits)
```

- The digits in the **significand** (the part before the ‘e’) are called the significant digits. The number of significant digits defines a number’s **precision**. The more digits in the significand, the more precise a number is.
- The precision of a floating point number defines how many significant digits it can represent without information loss.

Floating point range

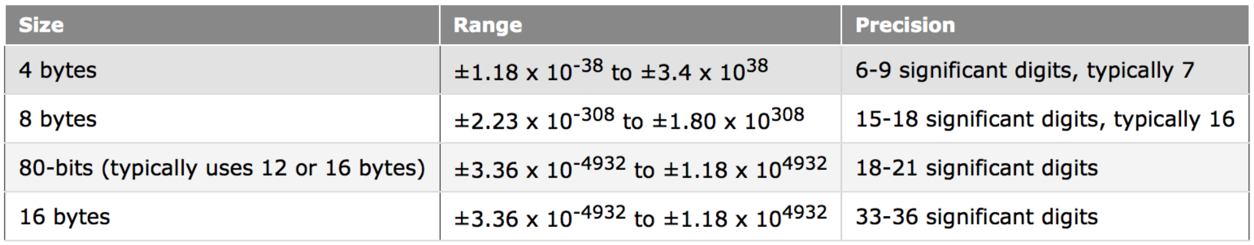
There are 3 types of floating point data types:

```
float fValue;
double dValue;
long double ldValue;
```



Type	Minimum Size	Typical Size
float	4 bytes	4 bytes
double	8 bytes	8 bytes
long double	8 bytes	8, 12, or 16 bytes

Assuming IEEE 754 representation:



Size	Range	Precision
4 bytes	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	6-9 significant digits, typically 7
8 bytes	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	15-18 significant digits, typically 16
80-bits (typically uses 12 or 16 bytes)	$\pm 3.36 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$	18-21 significant digits
16 bytes	$\pm 3.36 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$	33-36 significant digits

- **float** (32bits/4bytes) : 1 bit for the sign, (8 bits for the exponent, and 23* for the value), i.e. float has 7 decimal digits of precision.
- **double** (64bits/8bytes) : 1 bit for the sign, 11 bits for the exponent, and 52* bits for the value), i.e. double has 15 decimal digits of precision.

Floating point precision

The following program shows std::cout truncating to 6 digtis

```
#include <iostream>
int main()
{
    std::cout << 9.87654321f << '\n'; // 9.87654
    std::cout << 987.654321f << '\n'; // 987.654
    std::cout << 987654.321f << '\n'; // 987654
    std::cout << 9876543.21f << '\n'; // 9.87654e+006
    std::cout << 0.0000987654321 ;   // 9.87654e-005

    return 0;
}
```

- Note that each of these digits have only 6 significant digits
- Due to the latter point we won't be able to say if a value has been output with float or double precision
- We can override the default precision that std::cout shows by using the std::setprecision() function. We now see a clear distinction between float and double.

  ```
  #include <iostream>
  #include <iomanip> // for std::setprecision()
  int main()
  {
      std::cout << std::setprecision(16); // show 16 digits of precision
      std::cout << 3.33333333333333333333333333333333333333f <<'\n'; // f suffix means float
      std::cout << 3.33333333333333333333333333333333333333 << '\n'; // no suffix means double
      return 0;
  }
  3.333333253860474
  3.333333333333334
  ```

> Favor double over float unless space is at a premium, as the lack of precision in a float will often lead to inaccuracies

Rounding errors

From the example below we see that the output of f has been tweaked, this is called a r**ounding error**.

```
#include <iostream>
#include <iomanip> // for std::setprecision()

int main()
{
```

```
    float f { 123456789.0f };          // f has 10 significant digits
    std::cout << std::setprecision(9); // to show 9 digits in f
    std::cout << f << '\n';            // 123456792
    return 0;
}
```

- Floating point numbers are tricky to work with due the way they are stored. For example, 0.1 is represented by the infinite sequence: 0.00011**0011**00110011..., because of which we'll run into precision problems.

  ```
  #include <iostream>
  #include <iomanip> // for std::setprecision()

  int main()
  {
      double d{0.1};
      std::cout << d << '\n';              // 0.1
      std::cout << std::setprecision(17);
      std::cout << d << '\n';              // 0.10000000000000001
      return 0;
  }
  ```

  - On the first line cout prints 0.1 as expected, this behaviour however is not replicated for the next cout statement.
  - This is because double had to truncate the approximation due to limited memory, resulting in a number that has only 16 significant digits.
  - Rounding errors tend to make a number either slightly smaller or larger.

> Never use floating point numbers for financial or currency data.

NaN and Inf

- Inf represents infinity (positive or negative)
- Nan represents "Not a Number"

  ```
  #include <iostream>

  int main()
  {
      double zero {0.0};
      double posinf { 5.0 / zero };         // positive infinity
      std::cout << posinf << std::endl;     // inf

      double neginf { -5.0 / zero };        // negative infinity
      std::cout << neginf << std::endl;     // -inf

      double nan { zero / zero };           // not a number (mathematically invalid)
  ```

```
    std::cout << nan << std::endl;        // nan

    return 0;
}
```

4.9 - Boolean Values

Variables that can have only two possible values : True and False. They are initialized using **bool**.

```
bool b1 {true};
bool b2 {false};
b1 = false;
bool b3 {}; // default initializes to false
bool b1 { !true }; // b1 will be initialized with the value false
bool b2 { !false }; // b2 will be initialized with the value true
```

Printing boolean values

- Printing boolean variables would result in displaying **0** (for false) and **1** (for true)**.**
- To display false/true instead of 0/1 one can use **std::boolalpha**.

```
int main()
{
    std::cout << true << std::endl;  // true evaluates to 1
    std::cout << !true << std::endl; // !true evaluates to 0
    std::cout << std::boolalpha;     // print bools as true or false
    std::cout << true << std::endl;  // prints true
    std::cout << false << std::endl; // print false
```

Integer to bool conversion

- One can't initialize a boolean with an integer using uniform initialization (bool a {3};)
- However, if a conversion occurs, 0 is converted to false and any other integer to true;

```
int main()
{
  bool b1 {4};     // ERROR : const exp evaluates to 4 which cannot be narrowed to type 'bool'
  bool b1 = 4 ;    // copy initialization allows implicit conversion from int to bool
  std::cout << b1; // 1
```

Inputting and Returning bool values

- Using std::cin to accept bool values can sometimes leave programmers a bit confused.

```
int main()
{
  bool b {}; // default initialize to false (0)
  std::cout << "Enter a boolean value: ";  // enter : true
  std::cin >> b;
  std::cout << "You entered: " << b;       // output: 0
  return 0;
}
```

  This happens cause, std::cin only accepts two inputs for boolean variables: 0 and 1. Any other inputs cause std::cin to silently fail.

- Boolean values are often used a return value for functions to check if something is false or not

```
#include <iostream>

// returns true if x and y are equal, false otherwise
bool isEqual(int x, int y)
{
    return (x == y); // operator== returns true if x equals y, and false otherwise
}

int main()
{
    std::cout << "Enter an integer: ";
    int x{ 0 };
    std::cin >> x;                      // 4

    std::cout << "Enter another integer: ";
    int y{ 0 };
    std::cin >> y;                     // 4

    std::cout << std::boolalpha;

    std::cout << x << " and " << y << " are equal? ";
    std::cout << isEqual(x, y);       // true
    return 0;
}
```

  In the above example, isEqual is a **bool** function, which return a boolean variable and the std::boolalpha library converts this to false/true base on the 0/1 return value.

4.10 - Introduction to if statements

- A simple example

```
#include <iostream>

// returns true if x and y are equal, false otherwise
bool isEqual(int x, int y)
{
    return (x == y); // operator== returns true if x equals y, and false otherwise
}

int main()
{
    std::cout << "Enter an integer: ";
    int x {};
    std::cin >> x;

    std::cout << "Enter another integer: ";
    int y {};
    std::cin >> y;

    std::cout << std::boolalpha; // print bools as true or false

    if (isEqual(x, y))
        std::cout << x << " and " << y << " are equal\n";
    else
        std::cout << x << " and " << y << " are not equal\n";

    return 0;
}
```

- Dealing with non-boolean conditionals

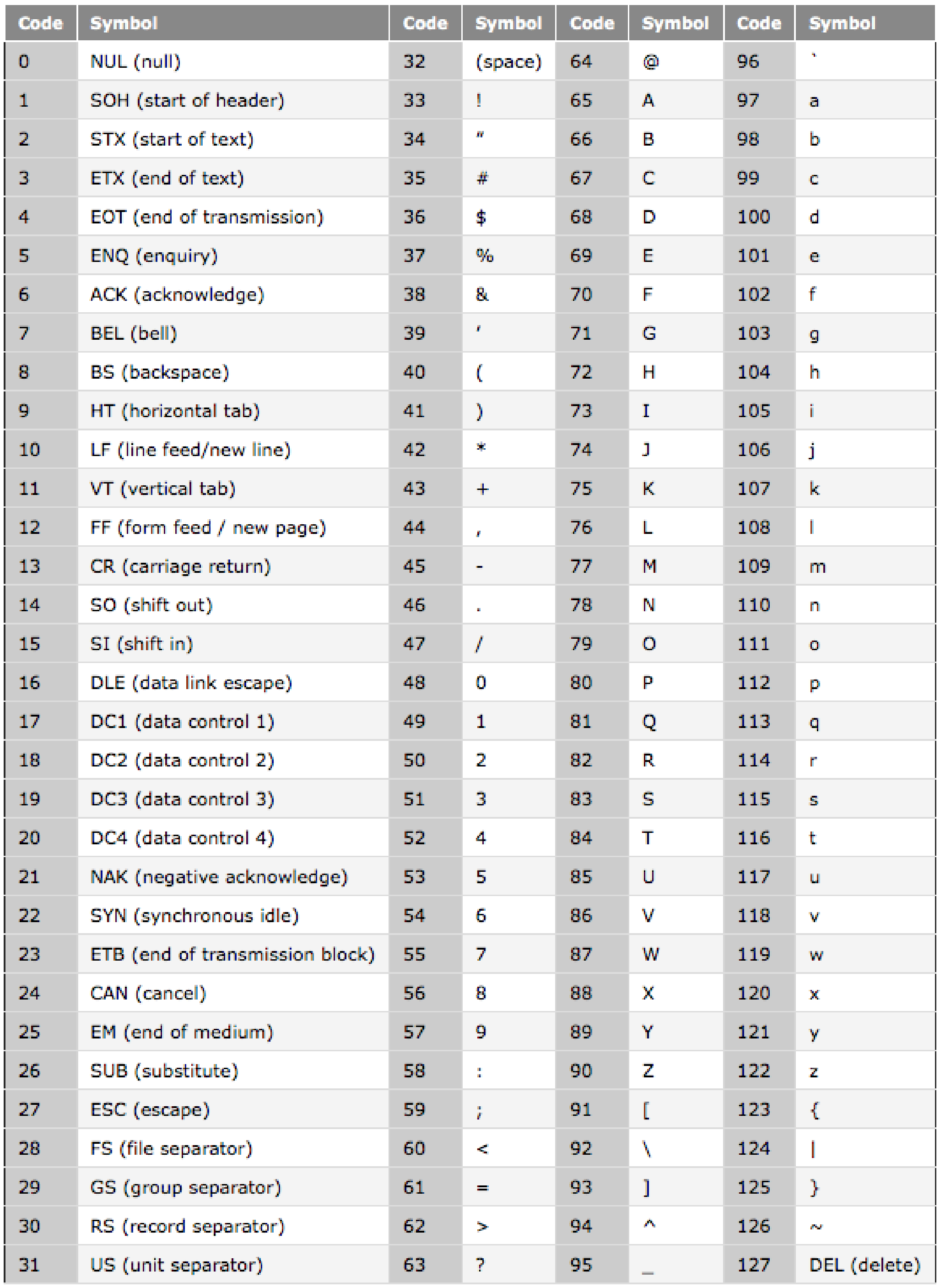
```
int main()
{
    if (4)
        std::cout << "hi";   // "hi is displayed as 4 is converted to bool true"
    else
        std::cout << "bye";

    return 0;
}
```

- Write a program that asks the user to enter a single digit integer. If the user enters a single digit that is prime (2, 3, 5, or 7), print “The digit is prime”. Otherwise, print “The digit is not prime”.

```
#include <iostream>
#include <stdio.h>
#include <list>

bool chkPrime(int a)
{
    std::list<int> listOfInts({2, 3, 7, 5});
    return (find(listOfInts.begin(), listOfInts.end(), a) != listOfInts.end());
    // or
    // return (a==2 || a==3 || a==5 || a==7);
}

int main()
{
    cout << "Enter a single digit number : ";
    int a{};
    cin >> a;

    if (chkPrime(a))
        cout << a << " is prime \n";
    else
        cout << a << " is not prime \n";


    return 0;

}
```

4.11 - Chars

- Char type is an **Integral type**, wherein the underlying value is stored as an integer and it's guaranteed to be 1-byte in size.
- Char value is interpreted as an ASCII - American Standard Code for Information Interchange -character.
- Below is a table of ASCII characters and their code
    - The characters go from 0-127, thus in essence only requiring 7-bits, but chars are assigned 8-bits to also take into consideration special characters from other languages.
    - Codes 0-31 are called **unprintable chars**, and are mostly used to perform formatting and control printers. Most of them are obsolete now.
    - Codes 32-127 are called printable characters, and are represented by numbers, characters and punctuations.



Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
0	NUL (null)	32	(space)	64	@	96	`
1	SOH (start of header)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	HT (horizontal tab)	41	)	73	I	105	i
10	LF (line feed/new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (form feed / new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (data control 1)	49	1	81	Q	113	q
18	DC2 (data control 2)	50	2	82	R	114	r
19	DC3 (data control 3)	51	3	83	S	115	s
20	DC4 (data control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of transmission block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL (delete)

Initializing chars

- Char variables can be initialized using char literals

```
char ch1{'a'}; // initialize with code point for 'a' (stored as integer 97)
std::cout << ch1; // cout prints a character
```

```
char ch2{ 97 }; // initialize with integer 97 ('a') (not preferred)
std::cout << ch2; // cout prints a character
```

- Avoid mixing char numbers with int numbers, the following two initializations are not the same

```
char ch{5}; // initialize with integer 5 (stored as integer 5)
char ch{'5'}; // initialize with code point for '5' (stored as integer 53)
```

Printing chars as int via type casting

- Traditionally, were one to convert char to int, the code would be as below

  ```
  int main()
  {
      char ch{97};
      int i(ch); // assign the value of ch to an integer
      std::cout << i; // print the integer value
      return 0;
  }
  ```

  This is however bulky and type cast can be used instead, which creates a value of one type from a value of another type.

- To convert between fundamental data-types, one can use a type cast called **static-cast**.

  ```
  int main()
  {
      char ch{ 'a' };
      std::cout << ch << '\n';                    \\ a
      std::cout << static_cast<int>(ch) << '\n';  \\ 97
      std::cout << ch << '\n';                    \\ a
      return 0;
  }
  ```

  The parameter **static_cast** evaluates as an expression. When a variable is passed, the variable is evaluated to produce its value which is converted to the new type. The variable is not affected by casting its value to a new typw.

Inputting, Char size, range and Default sign

- std::cin will let one enter multiple characters, but a variable defined as char will only hold 1 character, resulting in only the first input character being extracted.
- Char defined in C++ is always 1 byte of size

- By default char is signed or unsigned (usually signed).
- Sign need not be specified were char to hold ASCII cchars (both signed and unsigned can hold values in the range of 0-127).
- If one is using char to store small integers **(only to be used to optimize space usage)**, one should mention if it's signed or unsigned.

Escape sequences

- Characters in C++ that have special meanings. It usually beings with a '\' followed by a letter or a number: eg: "\n".

Name	Symbol	Meaning
Alert	\a	Makes an alert, such as a beep
Backspace	\b	Moves the cursor back one space
Formfeed	\f	Moves the cursor to next logical page
Newline	\n	Moves cursor to next line
Carriage return	\r	Moves cursor to beginning of line
Horizontal tab	\t	Prints a horizontal tab
Vertical tab	\v	Prints a vertical tab
Single quote	\’	Prints a single quote
Double quote	\”	Prints a double quote
Backslash	\\	Prints a backslash.
Question mark	\?	Prints a question mark. No longer relevant. You can use question marks unescaped.
Octal number	\(number)	Translates into char represented by octal
Hex number	\x(number)	Translates into char represented by hex number

```
int main()
{
    std::cout << "\"This is quoted text\"\n";
    std::cout << "This string contains a single backslash \\\n";
    std::cout << "6F in hex is char '\x6F'\n";
    return 0;
}
```

- Stand alone chars are always put in single quotes, eg : char a{'z'};

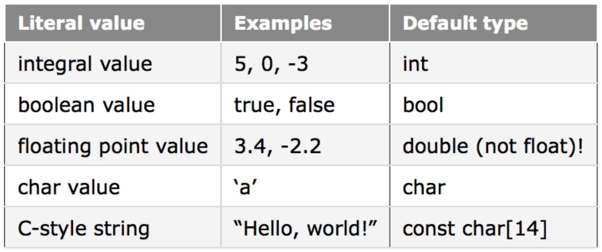
4.12 - Literals

- In programming, a **constant** is a fixed value that may not be changed. C++ has two kinds of constants : **literal constants** and symbolic constants. Literal constants (or literals) are values inserted into the code. Eg:

```
return 5; // 5 is an integer literal
bool myNameIsAlex { true }; // true is a boolean literal
std::cout << 3.4; // 3.4 is a double literal
```

- Since their values can't be changed dynamically (need to change them manually and then recompile the program for the change to take effect) they are called constants.

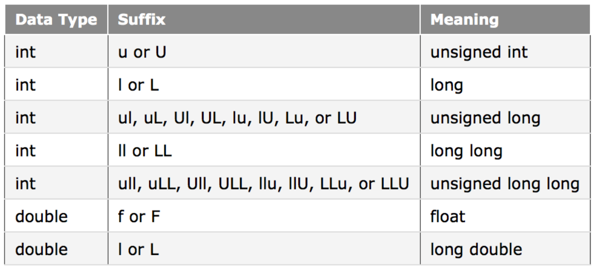
- Just like objects have a type, all literals have a type. The type is assumed from the value and format of the literal itself.



Literal value	Examples	Default type
integral value	5, 0, -3	int
boolean value	true, false	bool
floating point value	3.4, -2.2	double (not float)!
char value	‘a’	char
C-style string	“Hello, world!”	const char[14]

- If default type of the literal is not desired, one can change it by adding a suffix

```
unsigned int value1 { 5u }; // 5 has type unsigned
long value2 { 6L }; // 6 has type long
float f { 5.0f }; // 5.0 has type float
```



Data Type	Suffix	Meaning
int	u or U	unsigned int
int	l or L	long
int	ul, uL, Ul, UL, lu, lU, Lu, or LU	unsigned long
int	ll or LL	long long
int	ull, uLL, Ull, ULL, llu, llU, LLu, or LLU	unsigned long long
double	f or F	float
double	l or L	long double

- New programmers often get confused as to why the code below doesn't work as expected

```
float f { 4.1 }; // warning: 4.1 is a double suffix, not a float suffix

// Run the below program and verify the printed values
int main()
{
    std::cout << std::setprecision(16);
    std::cout << 4.12432423243234 <<endl <<4.12432423243234f <<endl;
    return 0;
}
```

Since 4.1 has no suffix, it's treated as a double literal and not a float. In C++, when a literal is defined, C++ doesn't care what's done with it (using to initialize a float variable). Hence 4.1 much be converted to float before assigning it to f.

- String literal work **strangely** in C++, so one can use them to print text with std::cout, but shouldn't try and assign them to variables or pass them to functions.
- Following are the ways how floating point literals can be declared

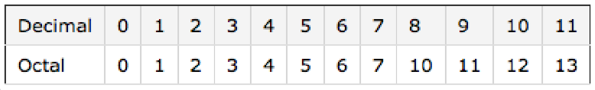
```
double pi { 3.14159 }; // 3.14159 is a double literal in standard notation
double avogadro { 6.02e23 }; // 6.02 x 10^23 is a double literal in scientific notation
double electron { 1.6e-19 }; // charge on an electron is 1.6 x 10^-19
```

Octal, hexadecimal and binary literals

In everyday life, we count using **decimal** numbers, with digits ranging from 0-9. Decimal is also called "base 10" (given that there are 10 possible digits) By default in C++, numbers are assumed to be decimal.

- In binary, there are only 2 digits, so it's called base 2.
- Octal is base 8, with digits ranging from 0-7. In C++ we can assign a octal literal to a variable using "**0**" prefix.

```
int main()
{
    int x{ 012 };   // 0 before the number means this is octal
    std::cout << x; // 10
    return 0;
}
```



Decimal	0	1	2	3	4	5	6	7	8	9	10	11
Octal	0	1	2	3	4	5	6	7	10	11	12	13

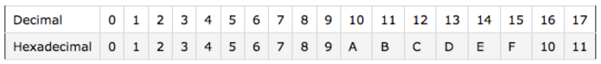
no 8 and 9, so we skip from 7 to 10

The output of the program will be 10 as C++ works with decimal numbers and the decimal equivalent of 12 octal is 10.

Octals are hardly used, stray away from it.

- Hexadecimal is to the base 16, and the count goes like this 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, .. . In C++ we can assign a hex literal to a variable using "**0x**" prefix.

```
int main()
{
    int x{ 0xF };   // 0x before the number means this is hexadecimal
    std::cout << x; // 15
    return 0;
}
```



Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11

- Given that there are 16 different values for a hexadecimal digit, one can say a single hex digit encompasses 4-bits

- Hexdecimal values are often used to represent address in memory due to their concise representation.

- In C++14 on can assign binary literal using **0b** prefix.

```
int main()
{
    int bin{};
    bin = 0b1;  // assign binary 0000 0001 to the variable
    bin = 0b11; // assign binary 0000 0011 to the variable
    bin = 0b1010; // assign binary 0000 1010 to the variable
    bin = 0b11110000; // assign binary 1111 0000 to the variable

    // Because long literals can be hard to read, C++14 also adds the
    // ability to use a quotation mark (‘) as a digit separator.
    int bin{ 0b1011'0010 };  // assign binary 1011 0010 to the variable
    long value{ 2'132'673'462 }; // much easier to read than 2132673462

    return 0;
```

- Printing decimal, octal and binary numbers

```
int main()
{
    int x { 12 };
    std::cout << x << '\n'; // decimal (by default)
    std::cout << std::hex << x << '\n'; // hexadecimal
    std::cout << x << '\n'; // now hexadecimal
    std::cout << std::oct << x << '\n'; // octal
//    std::cout << std::bin << x << '\n'; // binary
    std::cout << std::dec << x << '\n'; // return to decimal
    std::cout << x << '\n'; // decimal

    return 0;
}
```

- Printing binary is a little complicated beckoning the use of **std::bitset**, using which we can define a std::bitset variable and tell std::bitset how many bit would one like to store.

```
int main()
{
  // std::bitset<8> means we want to store 8 bits
  std::bitset<8> bin1{ 0b1100'0101 }; // binary literal for binary 1100 0101
  std::bitset<8> bin2{ 0xC5 }; // hexadecimal literal for binary 1100 0101

  std::cout << bin1 << ' ' << bin2 << '\n';
  std::cout << std::bitset<4>{ 0b1010 } << '\n'; // we can also print from std::bitset directly
  return 0;
}
```

4.13 - Const, constexpr, and symbolic constants

Sometimes during programming, it makes sense to define some constants - values that remain the same no matter what. In C++ this is done by adding the keyword **const** before or after the variable type.

```
const double gravity { 9.8 }; // preferred use of const before type
int const sidesInSquare { 4 }; // okay, but not preferred
```

- Doing so will prevent the value of the variable from changing during the course of the program.
- Additionally, when defining a constant, a value must be initialized too.
- Const variable can be initialized from other variables

  ```
  std::cout << "Enter age";
  int age;
  std::cin >> age;
  const int userAge {age};
  ```

- Const is often used with function arguments, ensuring that the function doesn't change the value of the passed argument.

  ```
  void printInteger(const int myValue)
  {
      std::cout << myValue;
  }
  ```

Run and compile time constants

- Runtime constants are those whose initialization values can only be resolved at runtime (when the program is running). **userAge** and **myValue** in the code snippets above are examples of such constants.
- Compile-time constants are those whose initialization values can be resolved at compile time (when the program is compiling). Variable **gravity** n the code snippets above is an example of such a constant.

```
#include <iostream>
#include <bitset>
#include <cstddef>

std::size_t getNumberOfBits()
{
    return 3;
}

int main()
{
    const std::size_t numberOfBits{ 3 }; // Compile-time constant

    std::bitset<numberOfBits> b{};

    const std::size_t otherNumberOfBits{ getNumberOfBits() }; // Run-time constant

    std::bitset<otherNumberOfBits> b2{}; // Error (otherNumberOfBits needs to have a v
                                         //        alue defined during compile, unfortunately,
                                         //        this will happen only during runtime

    return 0;
}
```

- For more specificity C++ introduced the keyword **constexpr**, which ensures that constant must be a compile-time constant
    - Any variable that shouldn't be modified after initialization and whose value is know at compile time should be declared a **constexpr.**
    - Any variable that shouldn't be modified after initialization and whose value is know at only at run time should be declared a **const.**

```
constexpr double gravity {9.9};  // okay, as gravity can be resolved at compile time
constexpr int sum { 4 + 5 };     // okay, as sum can be resolved at compile time
std::cout << "Enter age : ";
int age;
std::cin >> age;
contexpr int myAge( age };        // not okay, as age can't be resolved at compile time
```

- When we'd like to use a set of constants throughout different scripts, it makes sense to store them in a central location and use them wherever needed.

```
"constants.h"
ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace constants
{
    constexpr double pi { 3.14159 };
    constexpr double avogadro { 6.0221413e23 };
    constexpr double my_gravity { 9.2 }; // m/s^2 -- gravity is light on this planet
    // ... other related constants
}
#endif

"main.cpp"
#include "constants.h"
#include <iostream>

int main
{
    std::cout << "Enter a radius: ";
    int radius{};
    std::cin >> radius;

    double circumference { 2 * radius * constants::pi };
    std::cout << "The circumference is: " << circumference;

    return 0;
}
```