C++ Program Design
-- Variables and Fundamental Data Types

Junjie Cao @ DLUT

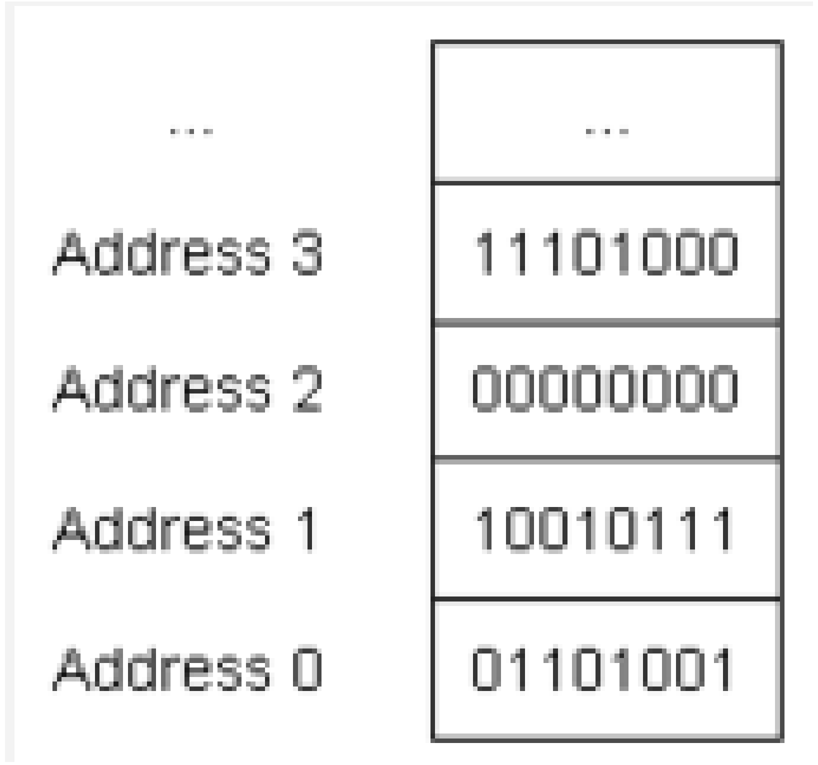
Summer 2016

http://jjcao.github.io/cPlusPlus

Fundamental variable definition, initialization, and assignment

variable definition, initialization, and assignment

- Bit: smallest unit of memory, hold a value of 0 or 1
- Memory is organized into sequential units called **addresses**: 0x007EFE94.
- **Byte:** smallest addressable unit of memory is a group of 8 bits

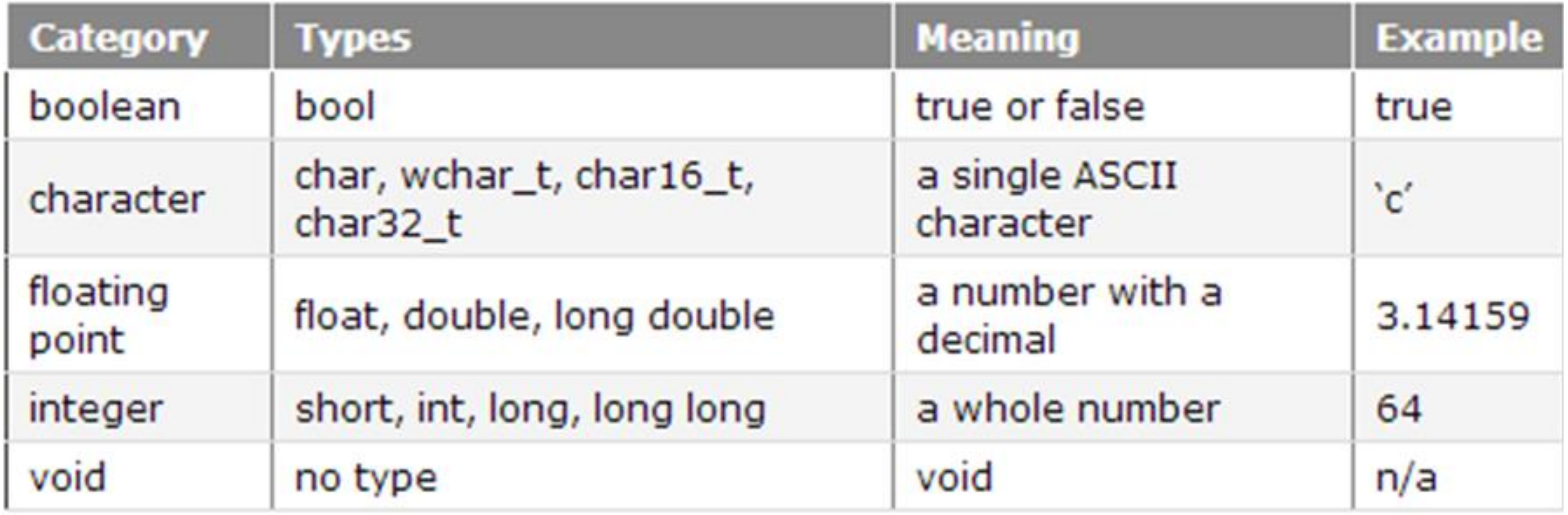


...	...
Address 3	11101000
Address 2	00000000
Address 1	10010111
Address 0	01101001

data type

- all data on a computer is just a sequence of bits
- **data type** to tell us how to interpret the contents of memory in some meaningful way
  - int x;
  - “the piece of memory that this variable addresses is going to be interpreted as a whole number”
- compiler and CPU take care of the details of encoding and decoding
  - x = 2;
  - int y(x);

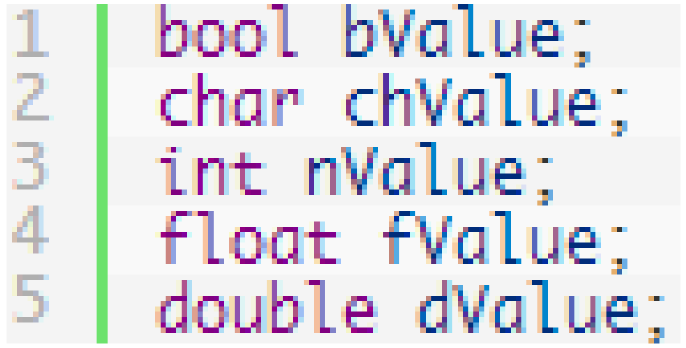
Fundamental data types



Category	Types	Meaning	Example
boolean	bool	true or false	true
character	char, wchar_t, char16_t, char32_t	a single ASCII character	‘c’
floating point	float, double, long double	a number with a decimal	3.14159
integer	short, int, long, long long	a whole number	64
void	no type	void	n/a

Defining a variable

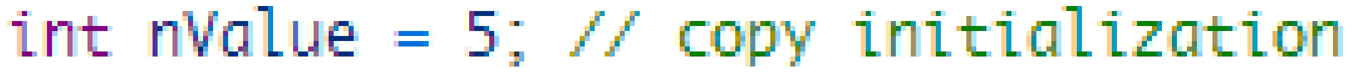
- type varName



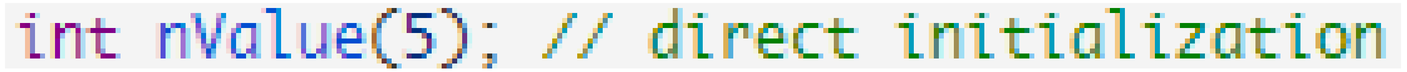
```
bool bValue;
char chValue;
int nValue;
float fValue;
double dValue;
```

Variable initialization

- **copy initialization vs. direct initialization**



```
int nValue = 5; // copy initialization
```



```
int nValue(5); // direct initialization
```

- Note = here is not = used to assign a value, once the variable has been created

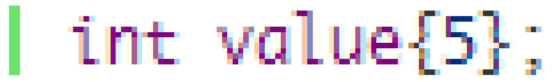
- Direct initialization: looks like a function call, but compiler knows

- Direct initialization can perform better than copy initialization for some data types

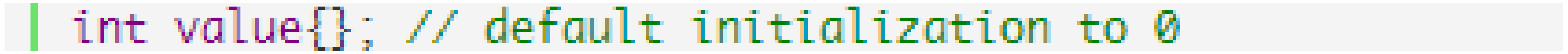
- *Rule: Favor direct initialization over copy initialization*

Uniform/list initialization in C++11

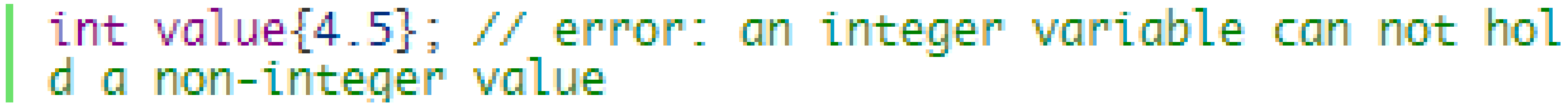
- copy initialization & direct initialization does not work for initialize a list of values, i.e. array



```
int value{5};
```



```
int value{}; // default initialization to 0
```



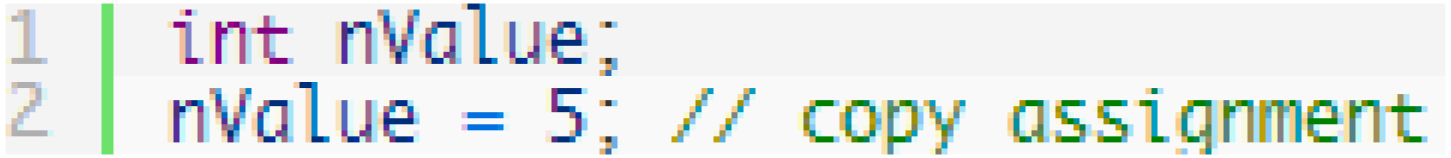
```
int value{4.5}; // error: an integer variable can not hol
d a non-integer value
```

- added benefit of disallowing “narrowing” type conversions.

- *Rule: If you’re using a C++11 compatible compiler, favor uniform initialization*

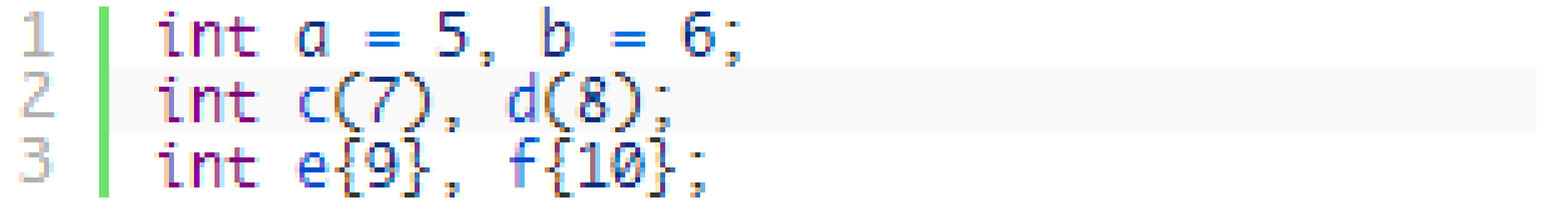
Variable assignment

- **copy assignment** (or assignment for short)
  - != copy initialization
  - When a variable is given a value after it has been defined

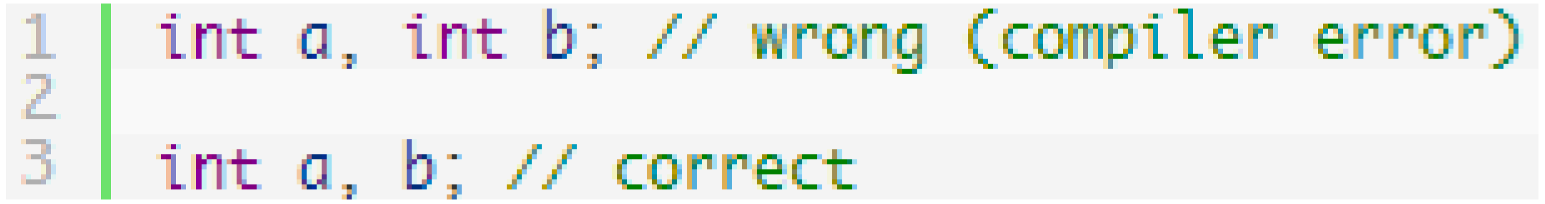


```
int nValue;
nValue = 5; // copy assignment
```

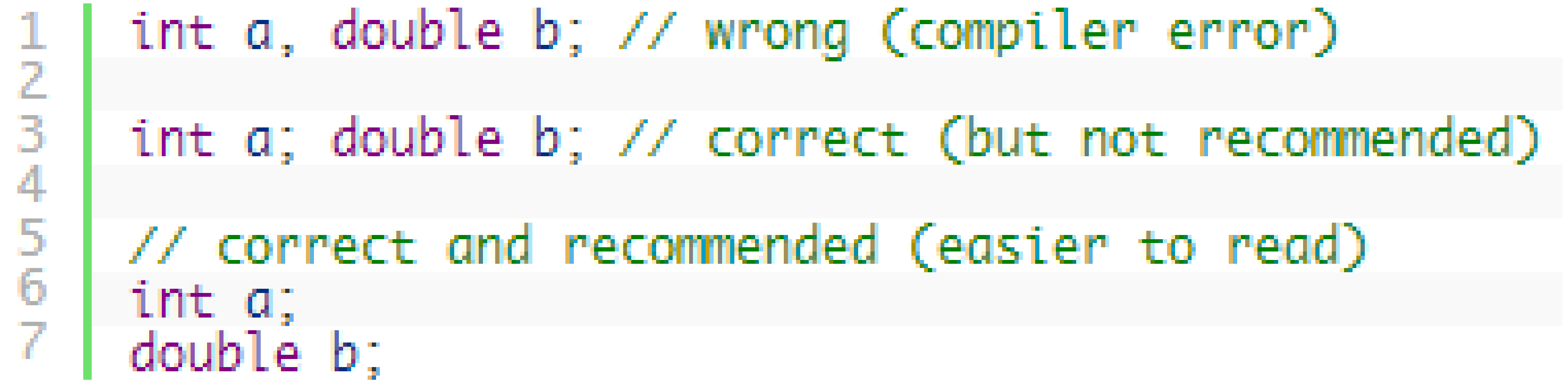
Defining multiple variables



```cpp
int a = 5, b = 6;
int c(7), d(8);
int e{9}, f{10};
```

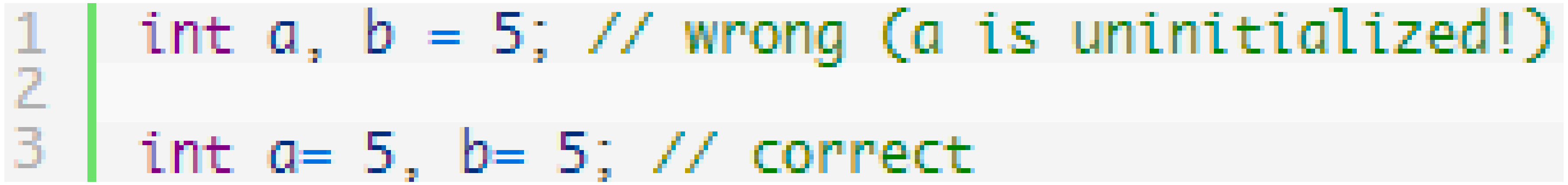


```cpp
int a, int b; // wrong (compiler error)

int a, b; // correct
```



```cpp
int a, double b; // wrong (compiler error)

int a; double b; // correct (but not recommended)

// correct and recommended (easier to read)
int a;
double b;
```

Defining multiple variables



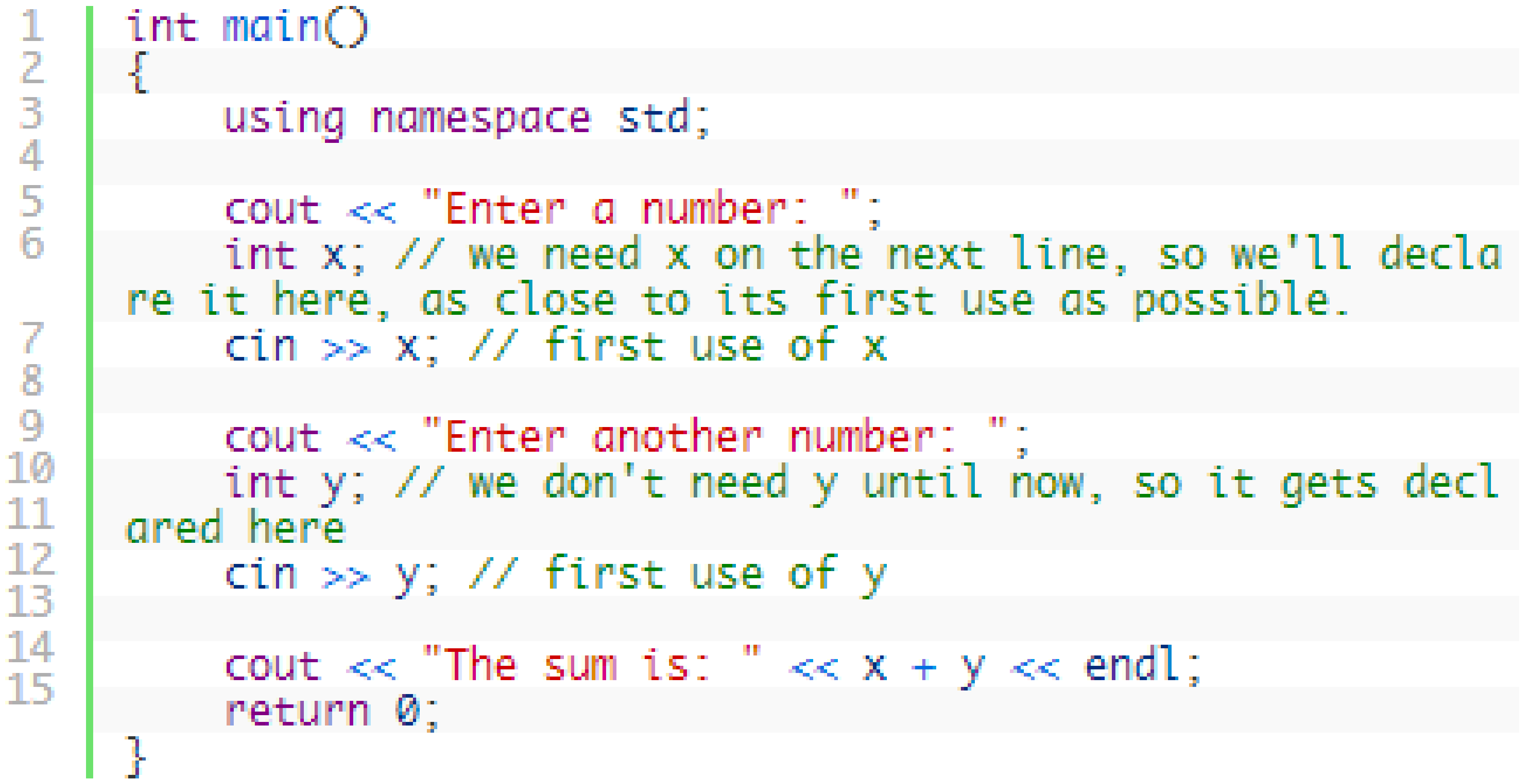
```
int a, b = 5; // wrong (a is uninitialized!)

int a= 5, b= 5; // correct
```

- Compiler may or may not complain
- “a” will be left uninitialized

- *Rule: Avoid defining multiple variables on a single line if initializing any of them*.

Where to define variables

- define all of the variables in a function at the top of the function
  - This style is now obsolete
- **define variables as close to the first use** of that variable as possible



```cpp
int main()
{
    using namespace std;

    cout << "Enter a number: ";
    int x; // we need x on the next line, so we'll declare it here, as close to its first use as possible.
    cin >> x; // first use of x

    cout << "Enter another number: ";
    int y; // we don't need y until now, so it gets declared here
    cin >> y; // first use of y

    cout << "The sum is: " << x + y << endl;
    return 0;
}
```

define variables as close to the first use

1. Easier to know: the variable is being used for …, etc input and/or output.
2. Make sure the variable does not affect anything above it
3. Reduces the likelihood of inadvertently leaving a variable uninitialized

- *Rule: Define variables as close to their first use as possible.*

Void

- means “no type”!

1. indicate that a function does not return a value:

```cpp
void writeValue(int x) // void here means no return value
{
    std::cout << "The value of x is: " << x << std::endl;
    // no return statement, because the return type is void
}
```

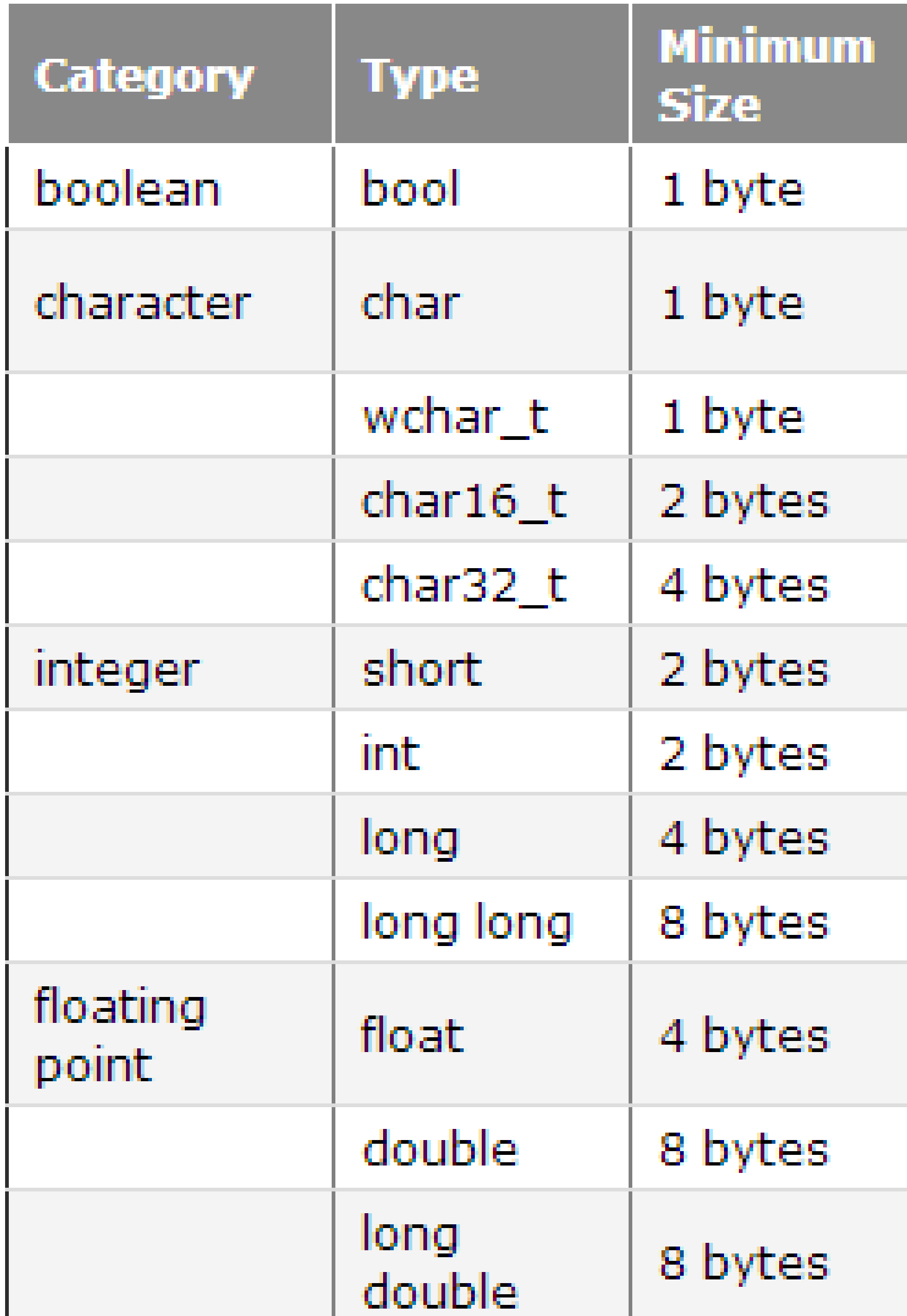
2. indicate that a function does not take any parameters:

```cpp
int getValue(void) // void here means no parameters
{
    int x;
    std::cin >> x;
    return x;
}
```

```cpp
void myVariable; // compiler error!
```

Variable sizes

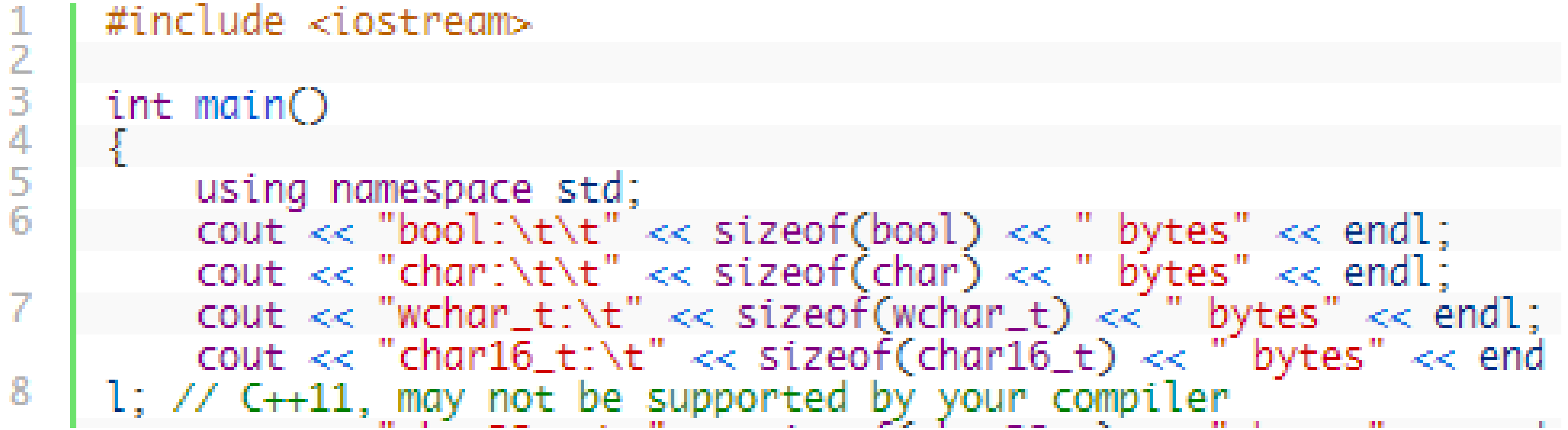
- a byte can store 2^8 (256) possible values.
- The size of the variable puts a limit on the amount of information it can store
- C++ guarantees that the basic data types will have a minimum size
- actual size of the variables may be different on your machine: **sizeof operator**



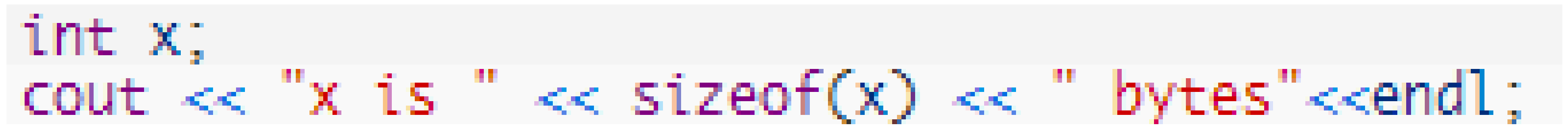
Category	Type	Minimum Size
boolean	bool	1 byte
character	char	1 byte
	wchar_t	1 byte
	char16_t	2 bytes
	char32_t	4 bytes
integer	short	2 bytes
	int	2 bytes
	long	4 bytes
	long long	8 bytes
floating point	float	4 bytes
	double	8 bytes
	long double	8 bytes

the sizeof operator

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



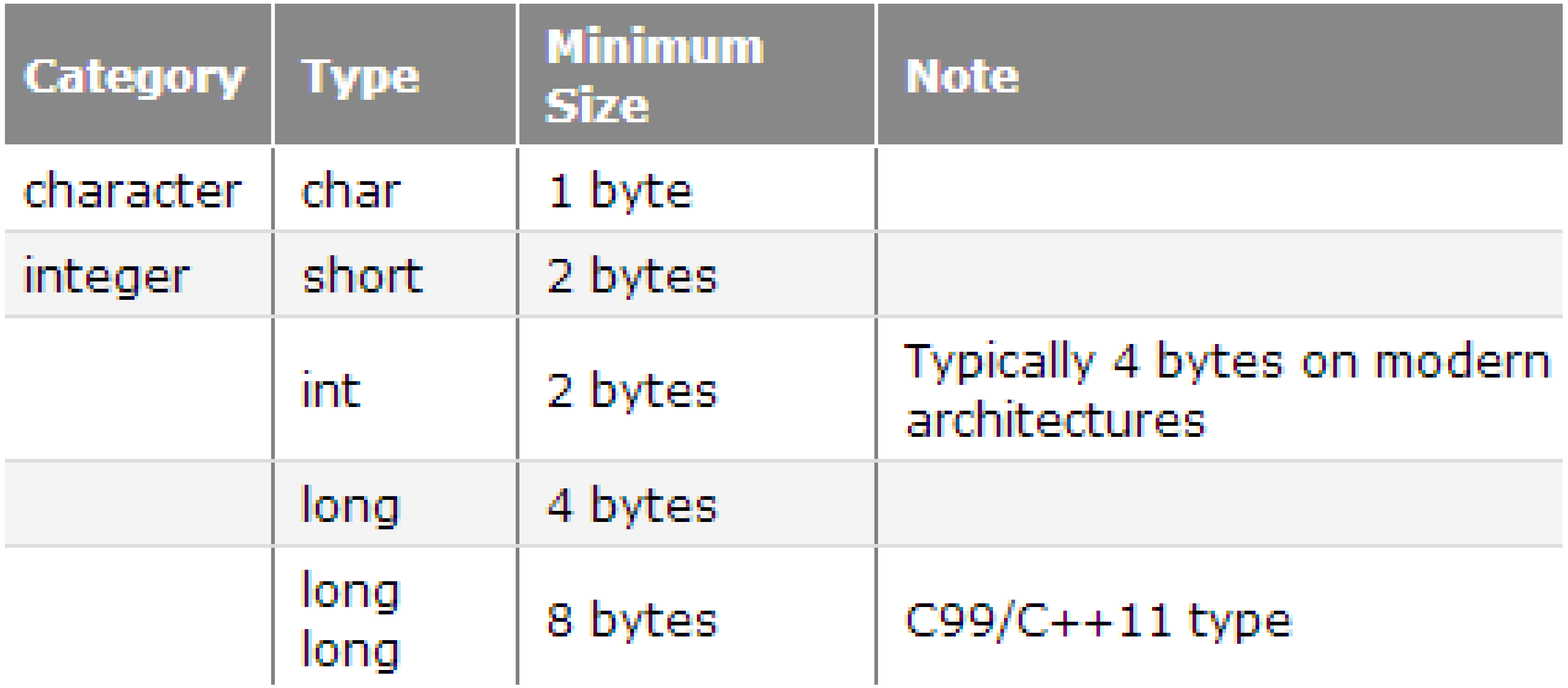
```
#include <iostream>

int main()
{
    using namespace std;
    cout << "bool:\t\t" << sizeof(bool) << " bytes" << endl;
    cout << "char:\t\t" << sizeof(char) << " bytes" << endl;
    cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes" << endl;
    cout << "char16_t:\t" << sizeof(char16_t) << " bytes" << end
l; // C++11, may not be supported by your compiler
```



```
int x;
cout << "x is " << sizeof(x) << " bytes"<<endl;
```

Integers

Integers

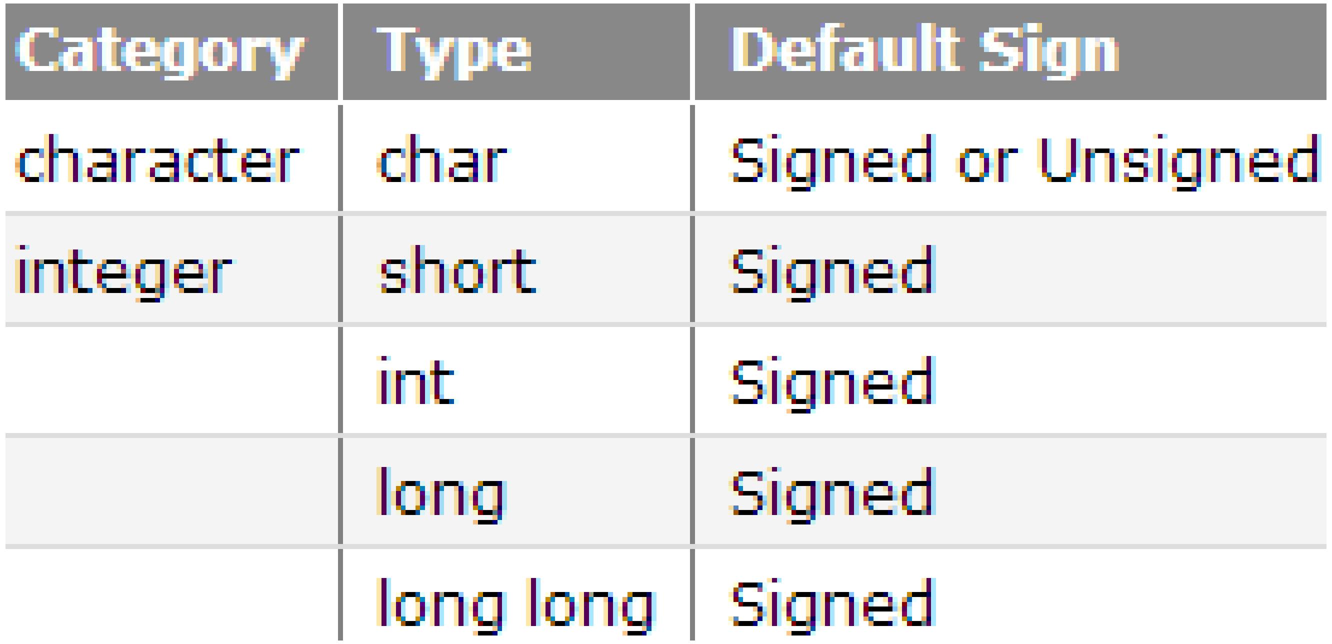


Category	Type	Minimum Size	Note
character	char	1 byte	
integer	short	2 bytes	
	int	2 bytes	Typically 4 bytes on modern architectures
	long	4 bytes	
	long long	8 bytes	C99/C++11 type

- Char is a special case, in that it falls into both the character and integer categories
- key difference: varying sizes -- the larger integers can hold bigger number

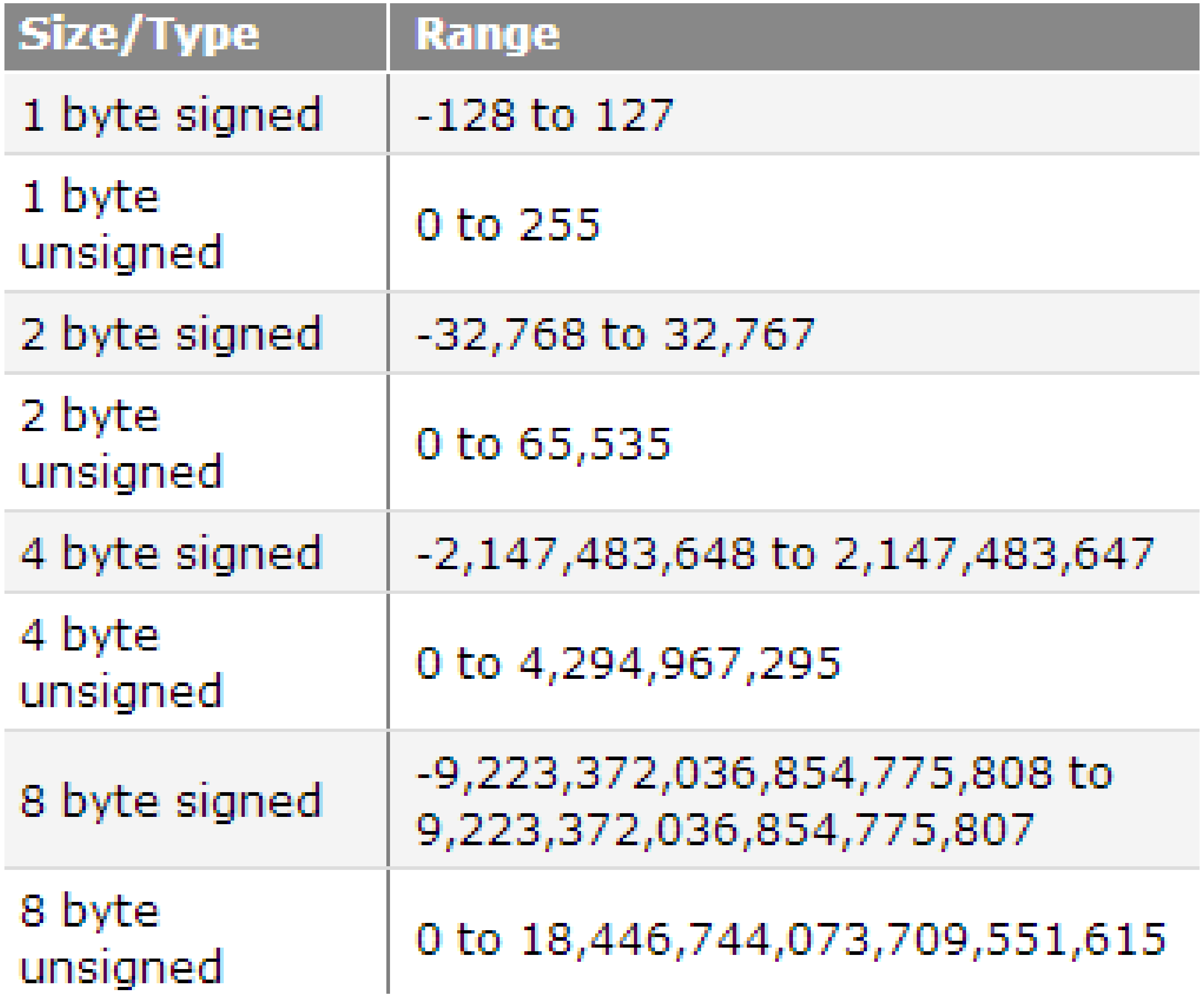
Integer ranges and sign

- A **signed** integer is a variable that can hold both negative and positive numbers.
  - A 1-byte signed integer has a range of -128 to 127
  - singed int i;
- An **unsigned** integer is one that can only hold positive values.
  - A 1-byte unsigned integer has a range of 0 to 255.
  - unsighed char c;
- **Default signs of integer**



Category	Type	Default Sign
character	char	Signed or Unsigned
integer	short	Signed
	int	Signed
	long	Signed
	long long	Signed

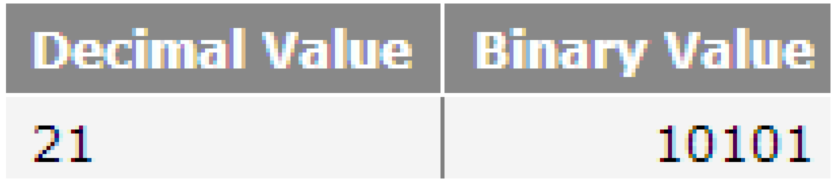
ranges and sign



Size/Type	Range
1 byte signed	-128 to 127
1 byte unsigned	0 to 255
2 byte signed	-32,768 to 32,767
2 byte unsigned	0 to 65,535
4 byte signed	-2,147,483,648 to 2,147,483,647
4 byte unsigned	0 to 4,294,967,295
8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
8 byte unsigned	0 to 18,446,744,073,709,551,615

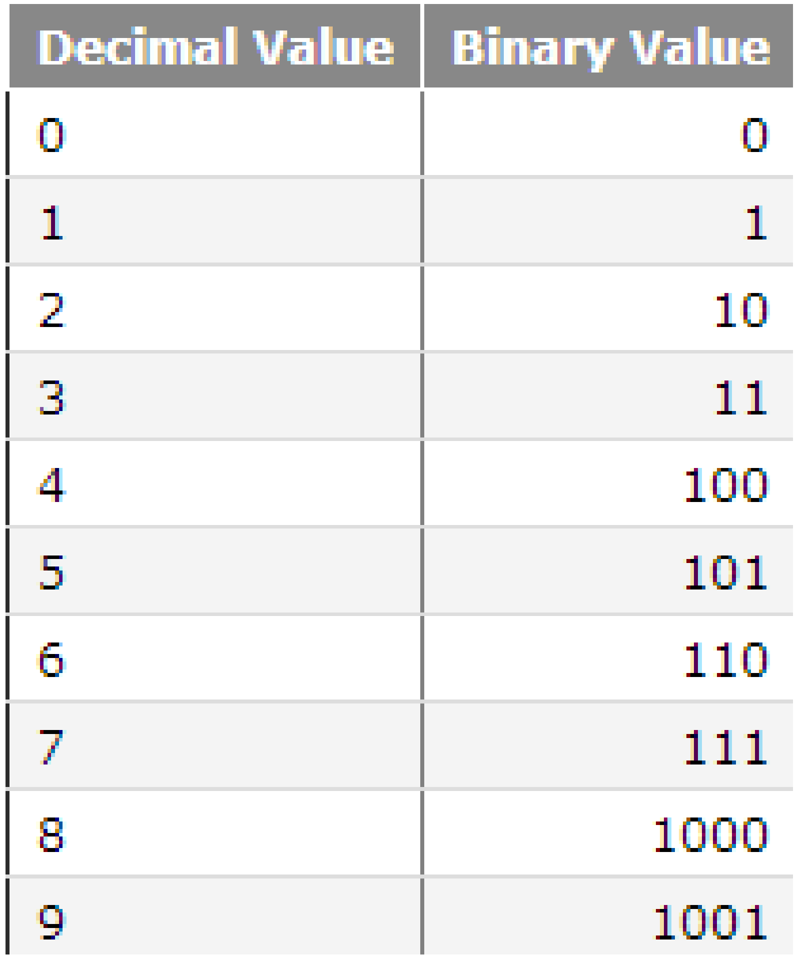
Overflow

- What happens if we try to put a number outside of the data type’s range into our variable?
- **Overflow** occurs when bits are lost because a variable has not been allocated enough memory to store them.
- put 21 in our 4-bit variable



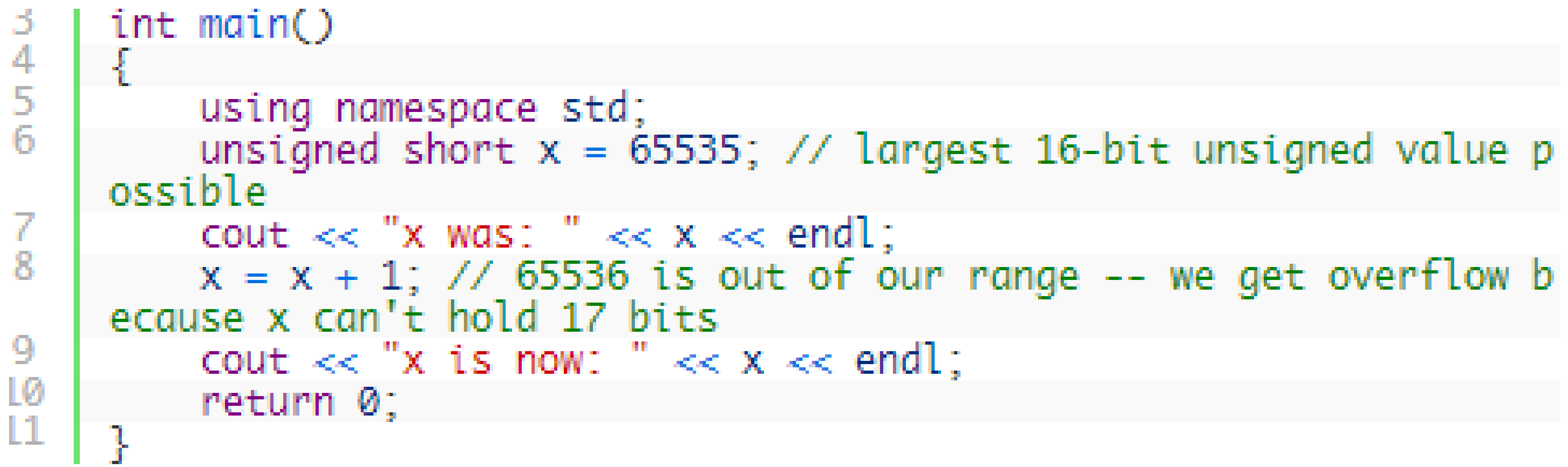
Decimal Value	Binary Value
21	10101

- 21 takes 5 bits (10101) to represent.
- The 4 rightmost bits (0101) go into the variable, and the leftmost (1) is simply lost.
- Our variable now holds 0101, which is the decimal value 5.



Decimal Value	Binary Value
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

Overflow



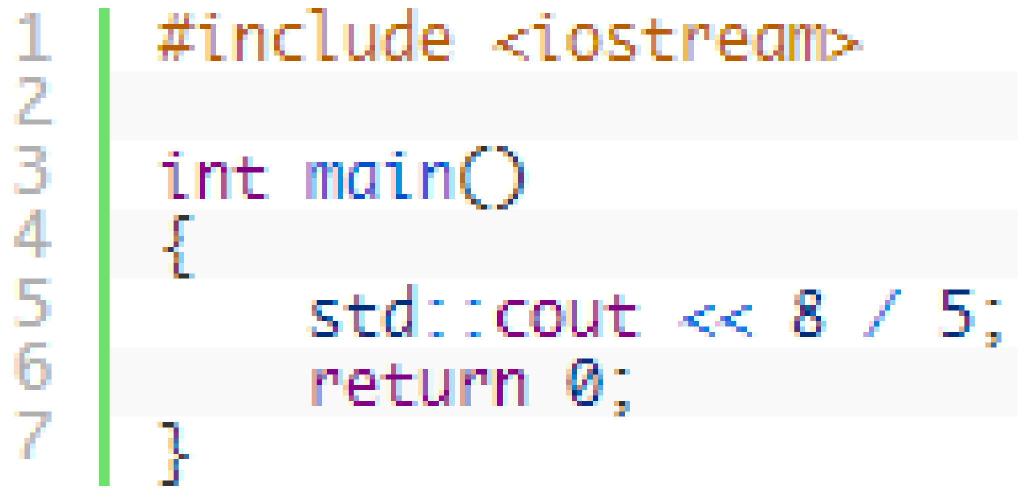
```
int main()
{
    using namespace std;
    unsigned short x = 65535; // largest 16-bit unsigned value possible
    cout << "x was: " << x << endl;
    x = x + 1; // 65536 is out of our range -- we get overflow because x can't hold 17 bits
    cout << "x is now: " << x << endl;
    return 0;
}
```

```
x was: 65535
x is now: 0
```

Rule: Do not depend on the results of overflow in your program.

Integer division

- When doing division with two integers, C++ produces an integer result.
- Since integers can’t hold fractional values, any fractional portion is simply **dropped** (not rounded!).



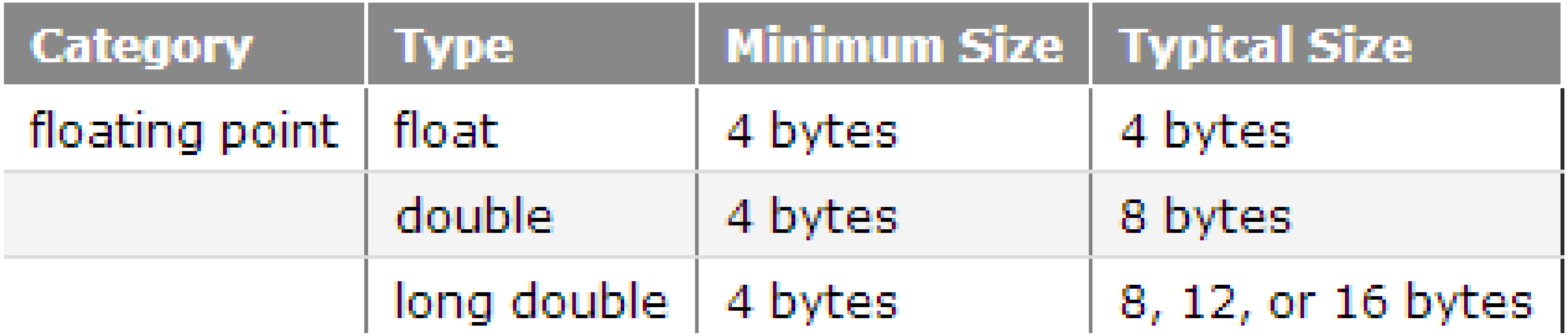
```cpp
#include <iostream>

int main()
{
	std::cout << 8 / 5;
	return 0;
}
```

- The above code produces: 1

- *Rule: Be careful when using integer division, as you will lose any fractional parts of the result*

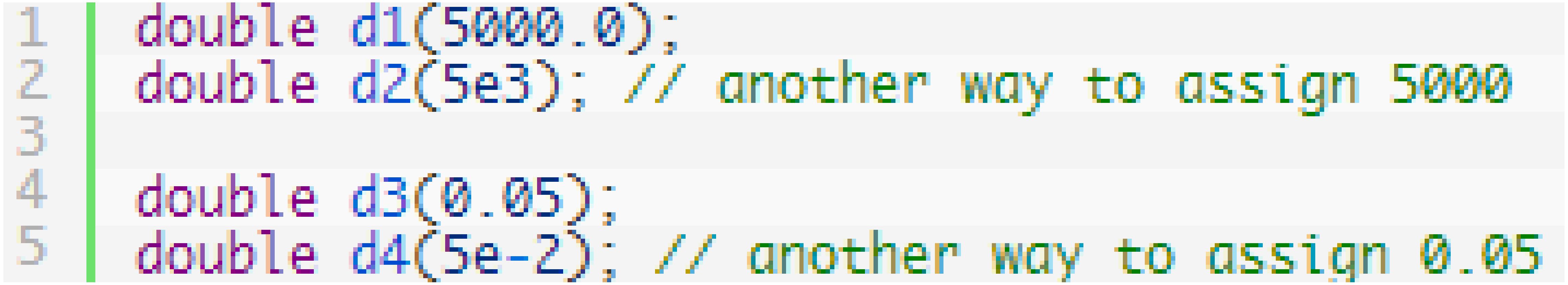
Floating point numbers

Floating point data



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

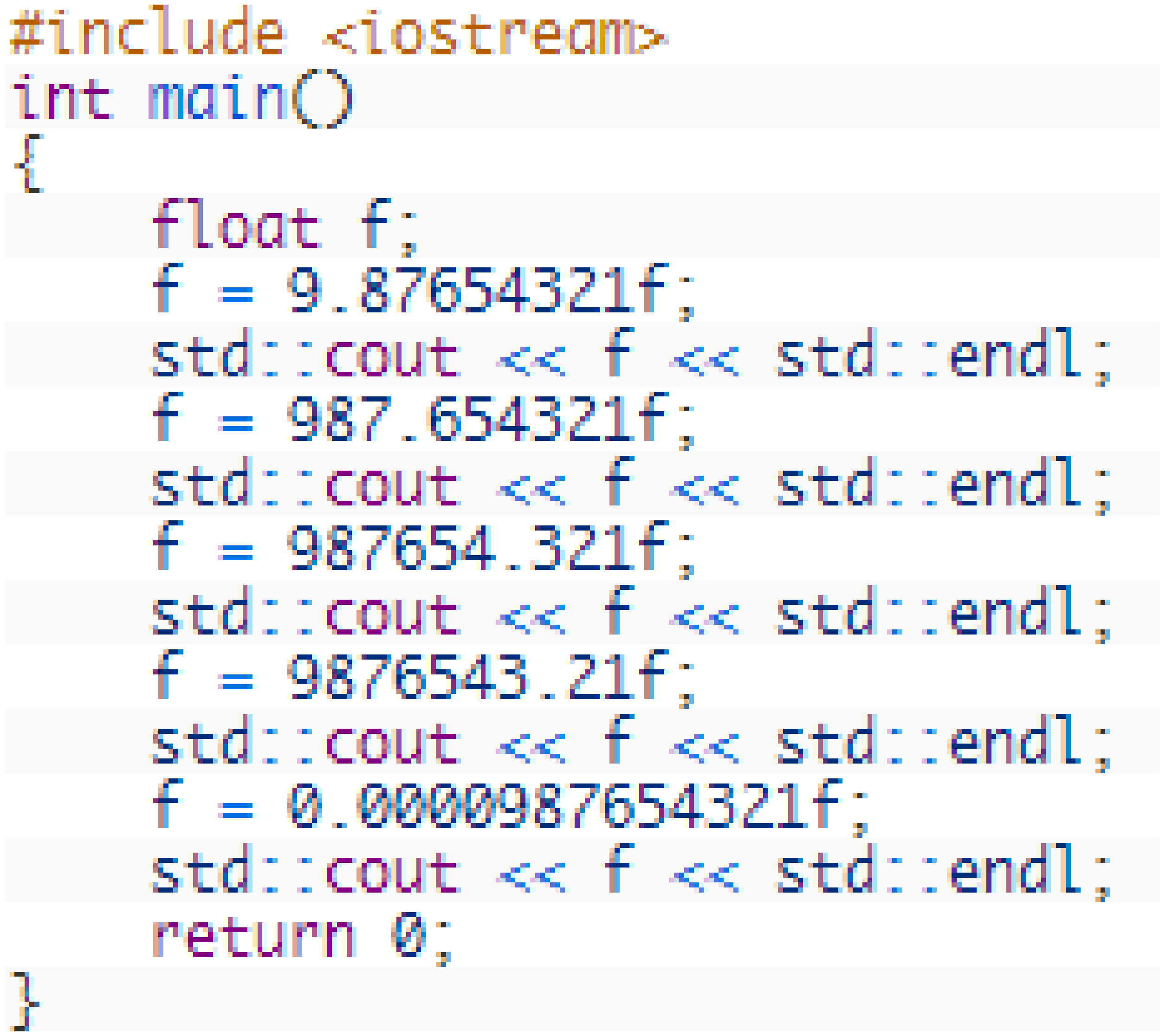
Scientific notation



```
double d1(5000.0);
double d2(5e3); // another way to assign 5000

double d3(0.05);
double d4(5e-2); // another way to assign 0.05
```

Precision and range

- The following program shows cout truncating to 6 digits:

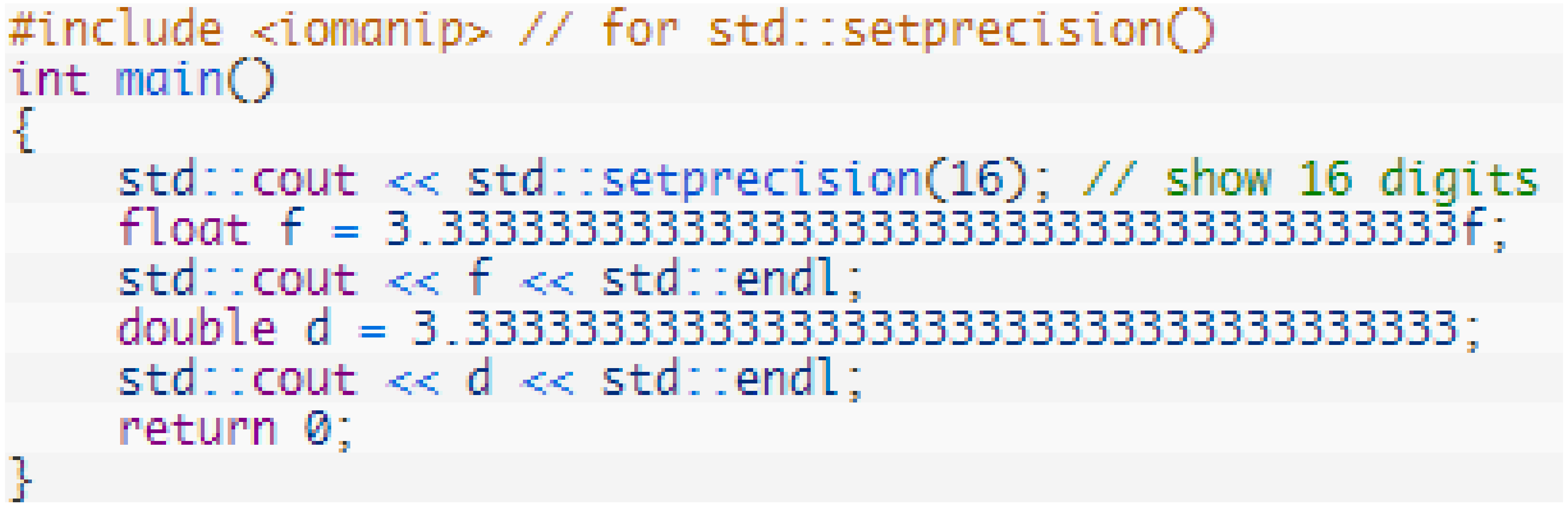


```
#include <iostream>
int main()
{
    float f;
    f = 9.87654321f;
    std::cout << f << std::endl;
    f = 987.654321f;
    std::cout << f << std::endl;
    f = 987654.321f;
    std::cout << f << std::endl;
    f = 9876543.21f;
    std::cout << f << std::endl;
    f = 0.0000987654321f;
    std::cout << f << std::endl;
    return 0;
}
```

- This program outputs:
- 9.87654
- 987.654
- 987654
- 9.87654e+006
- 9.87654e-005

Precision and range

- **std::setprecision()**



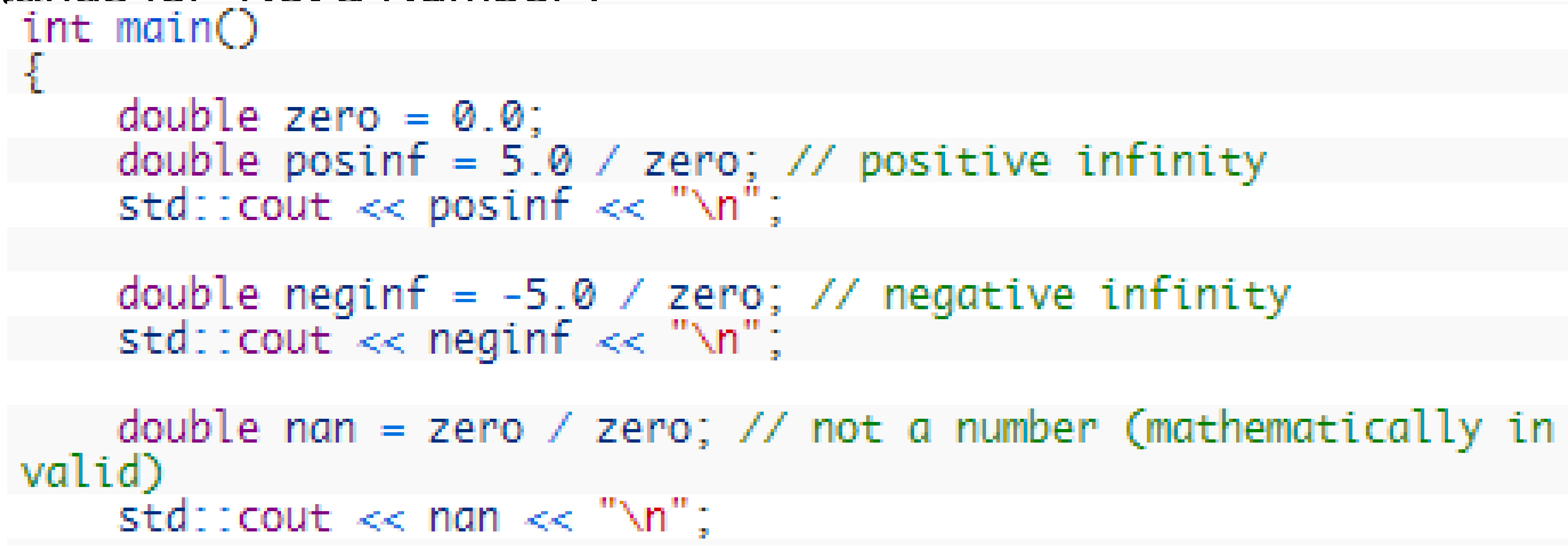
```
#include <iomanip> // for std::setprecision()
int main()
{
    std::cout << std::setprecision(16); // show 16 digits
    float f = 3.33333333333333333333333333333333333333f;
    std::cout << f << std::endl;
    double d = 3.3333333333333333333333333333333333333;
    std::cout << d << std::endl;
    return 0;
}
```

- Outputs:
  - 3.333333253860474
  - 3.333333333333334
- *Rule: Favor double over float unless space is at a premium, as the lack of precision in a float will often lead to inaccuracies.*

For more explanation, refer to http://www.learncpp.com/cpp-tutorial/25-floating-point-numbers/

NaN and Inf

- **Inf**, represents infinity.
- **NaN**, stands for “Not a Number”.



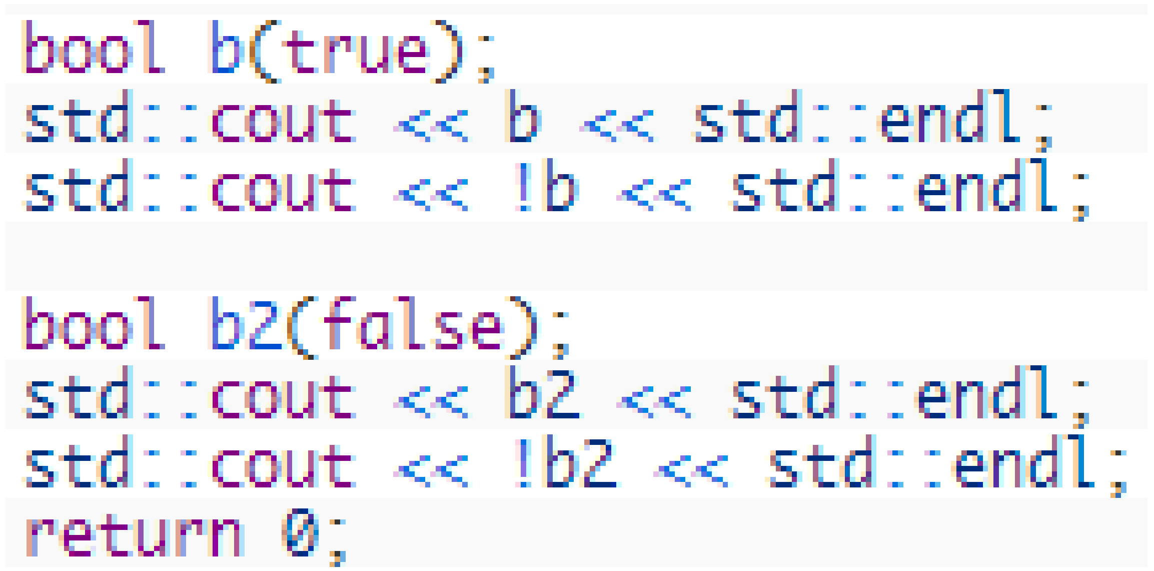
```
int main()
{
    double zero = 0.0;
    double posinf = 5.0 / zero; // positive infinity
    std::cout << posinf << "\n";

    double neginf = -5.0 / zero; // negative infinity
    std::cout << neginf << "\n";

    double nan = zero / zero; // not a number (mathematically in
valid)
    std::cout << nan << "\n";
```

- 1.#INF
- -1.#INF
- 1.#IND

Boolean values and if statements

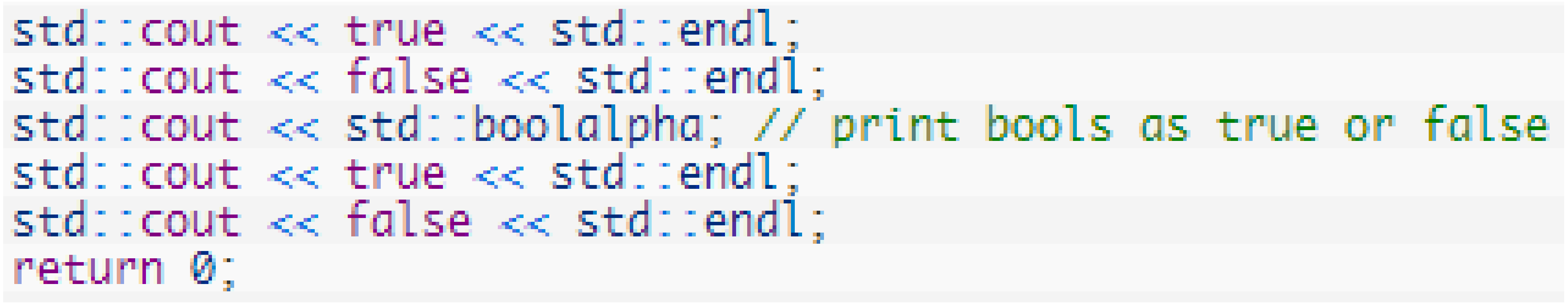
bool

- Boolean variables only have two possible values: true (1) and false (0).



```
bool b(true);
std::cout << b << std::endl;
std::cout << !b << std::endl;

bool b2(false);
std::cout << b2 << std::endl;
std::cout << !b2 << std::endl;
return 0;
```

std::boolalpha



```
std::cout << true << std::endl;
std::cout << false << std::endl;
std::cout << std::boolalpha; // print bools as true or false
std::cout << true << std::endl;
std::cout << false << std::endl;
return 0;
```

- This prints:
- 1
- 0
- true
- false

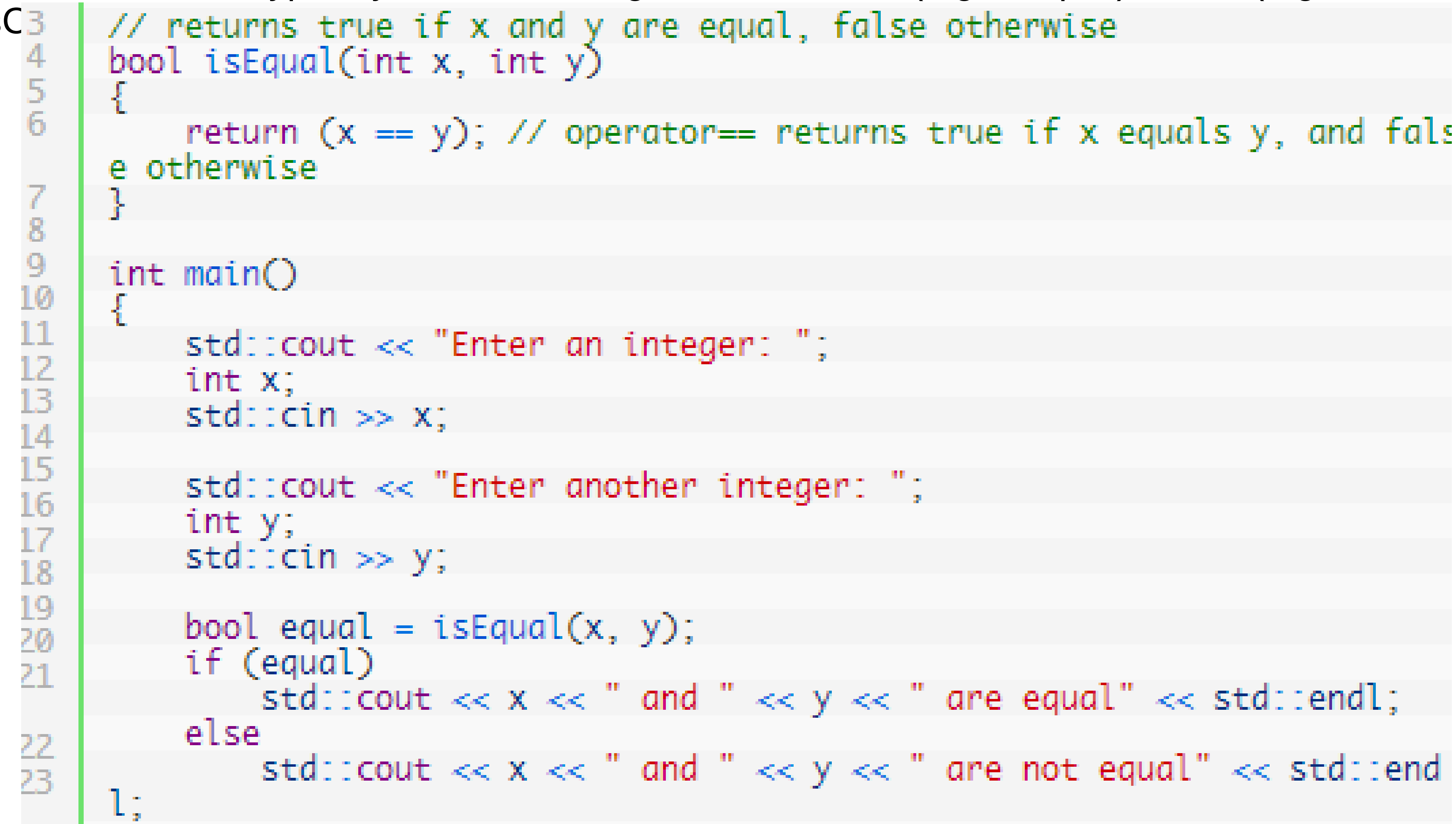
A first look at booleans in if-statements

```
if(condition)
{
        statementA1
        statementA2
        …
}
else
{
        statementB1
        statementB2
        …
}
```

```
if(condition)
        statementA1
else
        statementB1
```

- *condition* is evaluated. If …

- Boolean values are also useful as the return values for functions that check whether something is true or not.
- Such functions are typically named starting with the word *is*(e.g. isEqual) or *has* (e.g. hasC



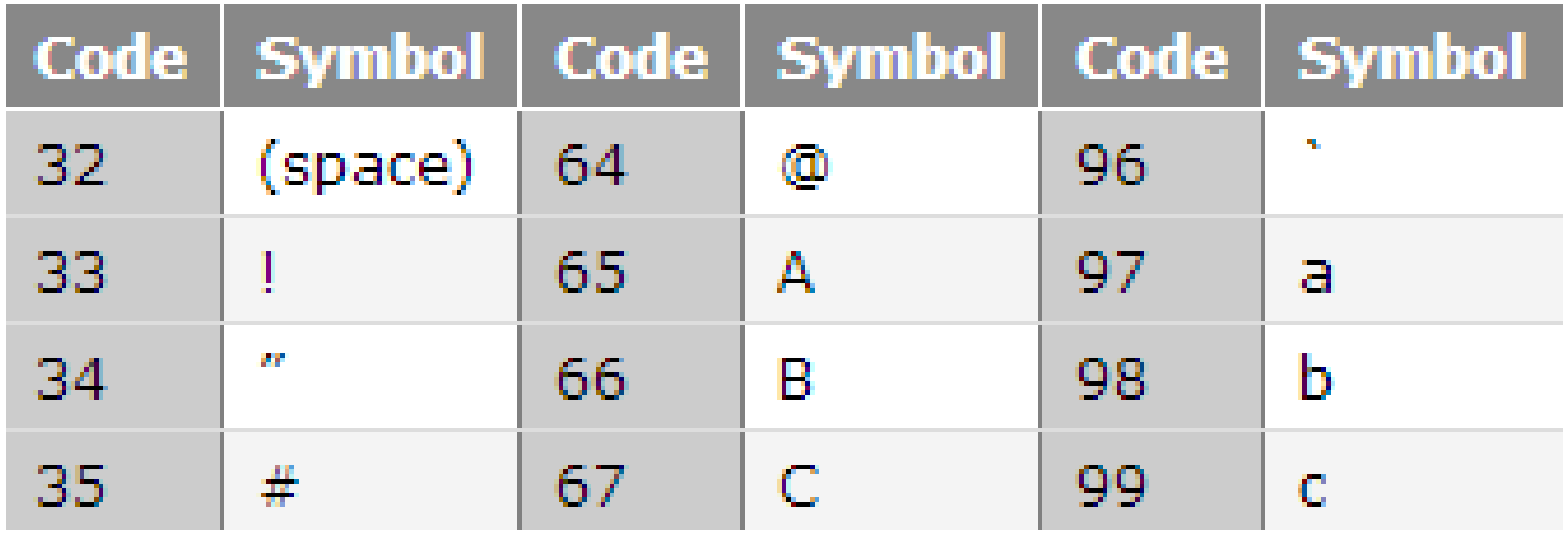
```cpp
// returns true if x and y are equal, false otherwise
bool isEqual(int x, int y)
{
    return (x == y); // operator== returns true if x equals y, and false otherwise
}

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

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

    bool equal = isEqual(x, y);
    if (equal)
        std::cout << x << " and " << y << " are equal" << std::endl;
    else
        std::cout << x << " and " << y << " are not equal" << std::endl;
```

Chars

Chars

- instead of interpreting the value of the char as an integer, the value of a char variable is typically *interpreted* as an ASCII character.

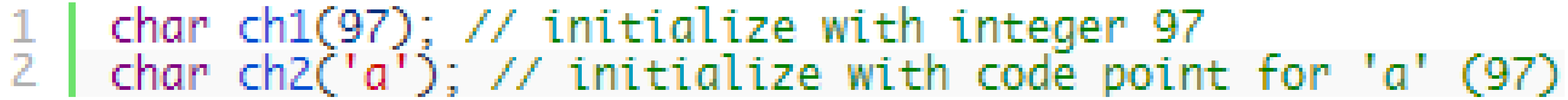
- **ASCII** stands for American Standard Code for Information Interchange, and it defines a particular way to represent English characters (plus a few other symbols) as numbers between **0 and 127**



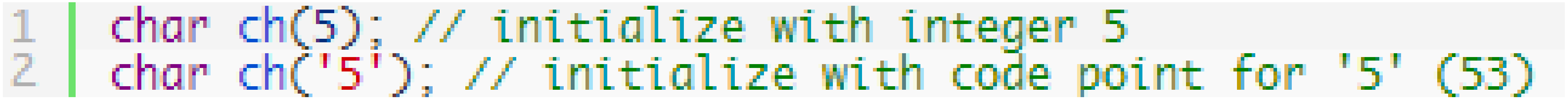
Code	Symbol	Code	Symbol	Code	Symbol
32	(space)	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c

Chars

- Initialization

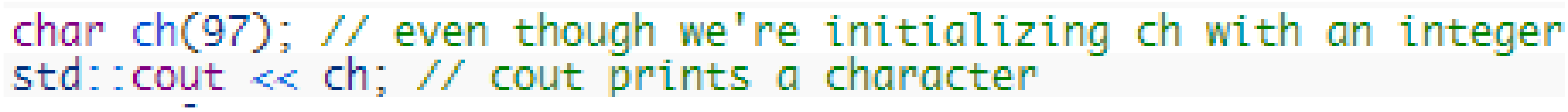


```cpp
char ch1(97); // initialize with integer 97
char ch2('a'); // initialize with code point for 'a' (97)
```



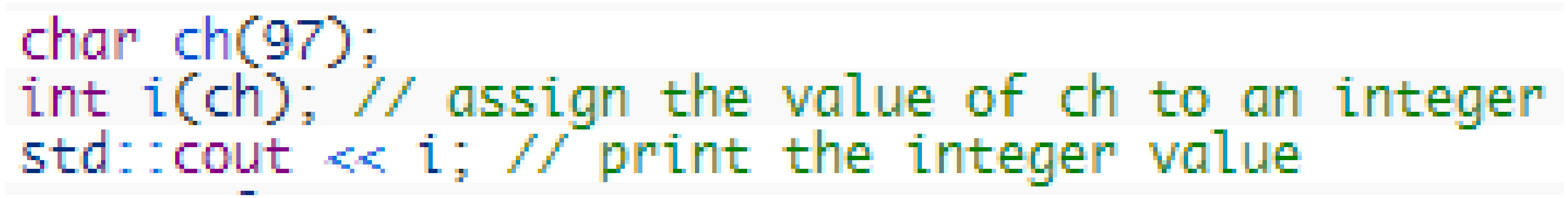
```cpp
char ch(5); // initialize with integer 5
char ch('5'); // initialize with code point for '5' (53)
```

- Printing



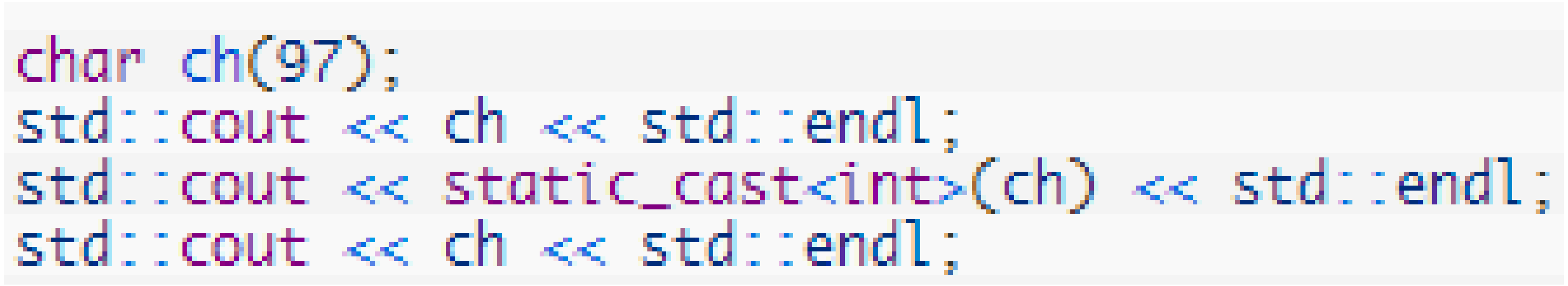
```cpp
char ch(97); // even though we're initializing ch with an integer
std::cout << ch; // cout prints a character
```

Printing chars as integers via type casting



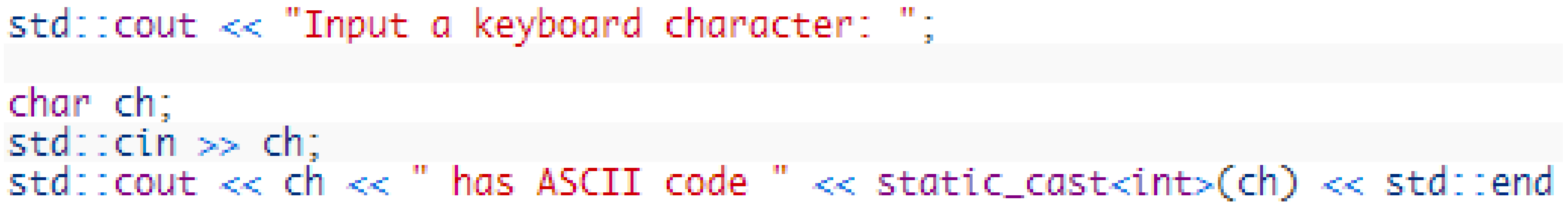
```
char ch(97);
int i(ch); // assign the value of ch to an integer
std::cout << i; // print the integer value
```

- **type cast**
  - static_cast<new_type>(expression)



```
char ch(97);
std::cout << ch << std::endl;
std::cout << static_cast<int>(ch) << std::endl;
std::cout << ch << std::endl;
```

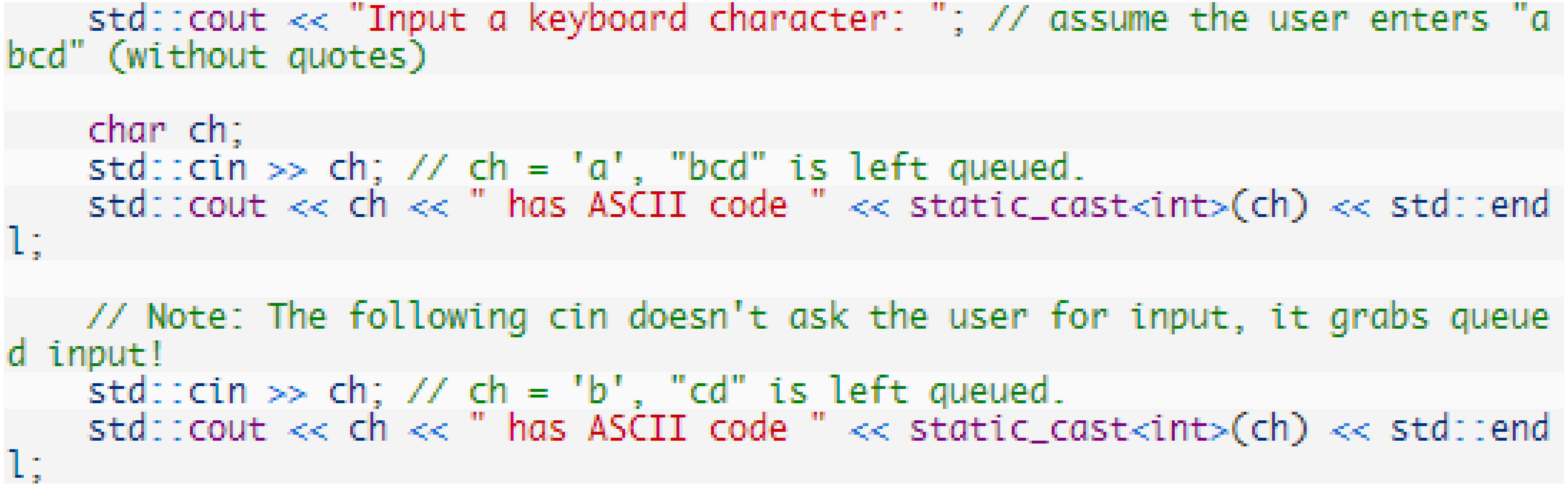
Inputting chars



```
std::cout << "Input a keyboard character: ";

char ch;
std::cin >> ch;
std::cout << ch << " has ASCII code " << static_cast<int>(ch) << std::end
```

- even though cin will let you enter multiple characters,
- ch will only hold 1 character.
- The rest of the user input is left in the input buffer that cin uses,
- and can be accessed with subsequent calls to cin.

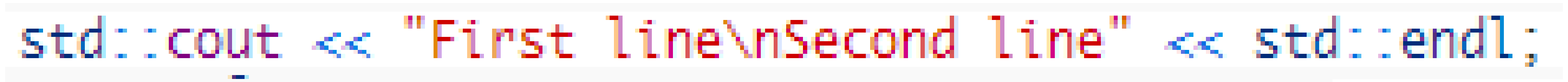
Inputting chars



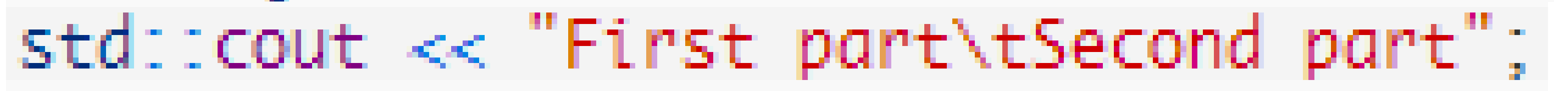
```
    std::cout << "Input a keyboard character: "; // assume the user enters "a
bcd" (without quotes)

    char ch;
    std::cin >> ch; // ch = 'a', "bcd" is left queued.
    std::cout << ch << " has ASCII code " << static_cast<int>(ch) << std::end
l;

    // Note: The following cin doesn't ask the user for input, it grabs queue
d input!
    std::cin >> ch; // ch = 'b', "cd" is left queued.
    std::cout << ch << " has ASCII code " << static_cast<int>(ch) << std::end
l;
```

Escape sequences

- some characters that have special meaning



```
std::cout << "First line\nSecond line" << std::endl;
```



```
std::cout << "First part\tSecond part";
```

- \’ prints a single quote
- \” prints a double quote
- \\ prints a backslash
- …

What about the other char types, wchar_t, char16_t, and char32_t?

- wchar_t should be avoided in almost all cases (except when interfacing with the Windows API).
- It has largely been deprecated.

- char16_t and char32_t were added to C++11 to provide explicit support for 16-bit and 32-bit Unicode characters

What’s the difference between putting symbols in single and double quotes?

- `char ch('56'); // a char can only hold one symbol`

- A **string** is a collection of sequential characters (and thus, a string can hold multiple symbols).
- `std::cout << "Hello, world!"; // "Hello, world!" is a string literal`

- string variables are a little more complex
- reserve discussion later.

Literals

Literal constants (usually just called "literals")

- C++ has two kinds of constants: literal, and symbolic
- Literals are literal numbers inserted into the code.
- `bool myNameIsAlex = true; // true is a boolean literal constant`
- `int x = 5; // 5 is an integer literal constant`
- `double electron = 1.6e-19; // charge on an electron is 1.6 x 10^-19`
- **'O'**
- **"Hello word"**
- **Escape Sequences for Nonprintable Characters:**
- **\n //newline**
- **\t //horizontal tab**

Octal and hexadecimal literals

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

- `int x = 012; // 0 before the number means this is octal`
- `int x = 0xFF; // 0x before the number means this is hexadecimal`

Magic numbers, and why they are bad

- `int maxStudents = numClassrooms * 30;`
- `setMax(30);`

- A **magic number** is a hard-coded literal (usually a number) in the middle of the code that does not have any context.
  - difficult to infer what a hard-coded number represents, unless there's a comment to explain it.
  - pose problems if the value needs to change
    - Is 30 in setMax the same with 30 in 1st line?
    - global search-and-replace?
- *Rule: Don't use magic numbers in your code*
- Solution: symbolic constants

symbolic constants

- Declaring a variable as const prevents us from changing its value

```
const double gravity { 9.8 };
gravity = 9.9; // not allowed, this will cause a compile error
```

- Defining a const variable without initializing it will also cause a compile error

```
const double gravity; // compiler error, must be initialized upon
definition
```

- const variables can be initialized from non-const values:

```
int age{6};
const int usersAge (age); // usersAge can not be changed
```

- Const is most useful (and most often used) with function parameters:

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

Constexpr

- C++ actually has two different kinds of constants
- **const**: **Runtime** constants are those whose initialization values can only be resolved at runtime

```
std::cout << "Enter your age: ";
int age; std::cin >> age;
const int usersAge (age); // usersAge can not be changed
```

- **Constexpr: Compile-time** constants are those whose initialization values can be resolved at compile-time

```
constexpr int sum = 4 + 5; // ok, the value of 4 + 5 can be resolved at compile-time
std::cout << "Enter your age: ";
int age; std::cin >> age;
constexpr int myAge = age; // not okay, age can not be resolved at compile-time
```

- *Rule: Any variable that should not change values after initialization should be declared as const (or constexpr).*

Symbolic constants

- it is a name given to a constant literal value
- two ways to declare symbolic constants in C++
- **Bad: Using object-like macros with a substitution parameter as symbolic constants**

```
#define MAX_STUDENTS_PER_CLASS 30
#define MAX_NAME_LENGTH 30

int max_students = numClassrooms * MAX_STUDENTS_PER_CLASS;
setMax(MAX_NAME_LENGTH);
```

  - #defined symbolic constants do not show up in the debugger
  - #defined values always have global scope => may have a naming conflict
- *Rule: Avoid using #define to create symbolic constants*

Symbolic constants

- **A better solution: Use const variables**

```
const int maxStudentsPerClass { 30 };
const int maxNameLength { 30 };
```

- *Rule: use const variables to provide a name and context for your magic numbers.*

Using symbolic constants throughout a program

1) Create a header file to hold these constants
2) Inside this header file, declare a namespace
3) Add all your constants inside the namespace (make sure they're const)
4) #include the header file wherever you need it

```
#ifndef CONSTANTS_H
#define CONSTANTS_H
namespace constants// define your own namespace to hold constants
{
    const double pi(3.14159);
    const double avogadro(6.0221413e23);
    const double my_gravity(9.2); // m/s^2 -- gravity is light on this planet
}#endif
```

```
#include "constants.h"
double circumference = 2 * radius * constants::pi;
```

Summary

- Variable
    - memory address, byte, bit
    - Initialization, assignment
    - sizeof, size and information
- Int, sign, overflow
- Float, range, precision
- Bool, char, ASII, Unicode
- Liberal constant, symbolic constant
    - #define
    - const variable

Quiz

- Why are symbolic constants usually a better choice than literal constants? Why are const symbolic constants usually a better choice than #defined symbolic constants?
- Pick the appropriate data type for a variable in each of the following situations. Be as specific as possible. If the answer is an integer, pick a specific integer type (e.g. int16_t) based on range. If the variable should be const, say so.

  a) The age of the user (in years)
  b) Whether the user wants color or not
  c) pi (3.14159265)
  d) The number of pages in a textbook
  e) The price of a stock in dollars (to 2 decimal places)
  f) How many times you've blinked since you were born (note: answer is in the millions)
  g) A user selecting an option from a menu by letter
  h) The year someone was born

Write the following program

- The user is asked to enter 2 floating point numbers (use doubles). The user is then asked to enter one of the following mathematical symbols: +, -, *, or /. The program computes the answer on the two numbers the user entered and prints the results. If the user enters an invalid symbol, the program should print nothing.

```
Enter a double value: 7
Enter a second double value: 5
Enter one of the following: +, -, *, or /: *
7 * 5 is 35
```

simulate a ball being dropped off of a tower

- To start, the user should be asked for the initial height of the tower in meters. Assume normal gravity (9.8 m/s^2), and that the ball has no initial velocity (the ball is not moving to start). Have the program output the height of the ball above the ground after 0, 1, 2, 3, 4, and 5 seconds. The ball should not go underneath the ground (height 0).

- Your program should include a header file named constants.h that includes a namespace called myConstants. In the myConstants namespace, define a symbolic constant to hold the value of gravity (9.8).

- Use a function to calculate the height of the ball after x seconds. The function can calculate how far the ball has fallen after x seconds using the following formula: distance fallen = gravity_constant * x_seconds^2 / 2

simulate a ball being dropped off of a tower

Learn C++

Sample output:

```
Enter the initial height of the tower in meters: 100
At 0 seconds, the ball is at height: 100 meters
At 1 seconds, the ball is at height: 95.1 meters
At 2 seconds, the ball is at height: 80.4 meters
At 3 seconds, the ball is at height: 55.9 meters
At 4 seconds, the ball is at height: 21.6 meters
At 5 seconds, the ball is on the ground.
```