HIGH-LEVEL PROGRAMMING 2

Intro to C++ and I/O

by Prasanna Ghali

Simplest C++ Program

In C++, you can specify function that takes no parameters using keyword `void` in parameter list or by using an empty parameter list

```
int main(void) {
  return 0;
}
```

If `return` statement not present in function `main`, C++ compiler will insert statement `return 0;`

Simplest C++ Program

- I like typing less, so my simplest C++ program will look like this:

```
int main() {
}
```

2nd Simplest Program

- C is a subset of C++
- C standard library is subset of C++ standard library

```
// ok: compiles with C++ compiler ...
#include <stdio.h>

int main() {
  printf("Hello World\n");
}
```

C++ Only: 2nd Simplest Program

Standard header declares global variables that control reading from and writing to standard streams `stdout`, `stdin`, and `stderr`

Namespaces are C++ mechanisms that introduce new scopes to avoid conflicts between names in large programs.
`std` is namespace for virtually all names in C++ standard library

```
#include <iostream>

int main() {
  std::cout << "Hello World\n";
}
```

`::` is *scope resolution operator*.
`std::cout` means "name `cout` in namespace scope `std`"

Global variable of class type `std::ostream` is instantiated at program startup and its purpose is to write characters to standard stream `stdout`

C90 is not Strongly Typed

- If C90/C11 compilers see undeclared function, they assume function takes unknown number of parameters and returns an `int`

```c
#include <stdio.h>

int main(void) {
  // ok: compiles with C compiler ...
  printf("3+7 == %d\n", add(3, 7));
  return 0;
}

int add(int lhs, int rhs) {
  return lhs+rhs;
}
```

C++ is Strongly Typed

- Unlike C, C++ requires all names be declared before their first use

```
#include <iostream>

int main() {
  // error: call to undeclared function add ...
  std::cout << "3+7 == " << add(3, 7) << "\n";
}

int add(int lhs, int rhs) {
  return lhs+rhs;
}
```

C++ Integer Types

- Microsoft compiler is 32-bit compiler while GCC and Clang are 64-bit compiler
- `sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`

Type Name	Number of Bytes
`char`	1
`short`	2
`int`	4
`long`	4/8
`long long`	8
`size_t` (declared in `<cstddef>`)	4/8

C++ Boolean Type

- Microsoft compiler is 32-bit compiler while GCC and Clang are 64-bit compiler

Type Name	Number of Bytes	Values
bool	1	true/false

```
int i = 10;
bool b = true;
std::cout << "b: " << b << " | "
          << std::boolalpha << b << "\n";
std::cout << "(i+b): " << (i+b) << "\n";
```

std::boolalpha [std::noboolalpha] is I/O stream manipulator that prints alphanumeric [integral] versions of true and false

C++ Integer Literals (1/2)

- Integer literals can be written as binary, octal, decimal, and hexadecimal values …

```
unsigned long int d = 42ul;
unsigned long int o = 052Ul;
unsigned long int x = 0x2auL;
unsigned long int b = 0b0010'1010UL;
```

0b prefix for integer literals is new since C++14

digit separator makes large values readable

C++ Integer Literals (2/2)

Literal	Type
`123`	`int`
`0173`	`int`
`0x7b`	`int`
`0b0111'1011`	`int`
`123u`	`unsigned int`
`123l` or `123L`	`long int`
`123u` or `123U`	`unsigned int`
`123ul` or `123UL`	`unsigned long int`
`123ll` or `123LL`	`long long int`
`123ull` or `123ULL`	`unsigned long long int`

C++ Floating-Point Types

- Microsoft compiler is 32-bit compiler while GCC and Clang are 64-bit compiler

Type Name	Number of Bytes
`float`	4
`double`	8
`long double`	8/16

C++ Floating-Point Literals

Literal	Type
`123.`	`double`
`123.f` or `123.f`	`float`
`123.l` or `123.L`	`long long double`
`1.23e2`	`double`
`3.141'592'653'590`	`double`

Querying Properties of Arithmetic Types

- C++ standard library provides way to query properties of arithmetic types

```
#include <iostream>
#include <limits>

std::cout << "min int: " << std::numeric_limits<int>::min() << "\n";
std::cout << “max int: " << std::numeric_limits<int>::max() << "\n";

std::cout << "lowest double: "
          << std::numeric_limits<double>::lowest() << "\n";
std::cout << "min double:    "
          << std::numeric_limits<double>::min() << "\n";
std::cout << "max double:    "
          << std::numeric_limits<double>::max() << "\n";
```

Relational Operator !=

- C++ inherits relational operators from C

C++ provides alternative keyword not_eq for != operator

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Logical Operators (1/2)

- C++ inherits these logical operators from C

Operator	Description
!	Not
&&	And
||	Or

high to low
precedence order

Logical Operators (2/2)

- In addition, C++ provides alternative spellings [in the form of keywords] for logical operators …

Operator Symbol	Operator Keyword	Description
!	not	Not
&&	and	And
||	or	Or

high to low precedence order

Lvalues and *Rvalues*: C (1/7)

- Original definition from C:
  - Every expression is an *lvalue* or an *rvalue*
  - *Lvalue* (short for *Left value*) expression can appear on left or right hand side of assignment expression

```
int a = 11, b = 22;

// a and b are both lvalues
a = b;      // ok
b = a;      // ok
a = a + b; // ok
```

Lvalues and *Rvalues*: C (2/7)

- Original definition from C:
  - Every expression is an *lvalue* or an *rvalue*
  - *Lvalue* (short for *Left value*) expression can appear on left or right hand side of assignment operator
  - *Rvalue* (short for *Right value*) expression can only appear on right hand side of assignment operator

```
double a = 0.0, b = 1.1, c = -2.0;
a = b+c;               // ok
b = std::abs(a*c); // ok
c = std::pow(b, std::abs(a)); // ok
30 = a*b; // error: rvalue on left of assignment
```

Lvalues and *Rvalues*: C (3/7)

- Addition of `const` type qualifier in C98 complicated definition
  - Every expression is an *lvalue* or an *rvalue*
  - *Lvalue* (short for *locater value*) is expression that refers to identifiable memory location
  - By exclusion, any non-*lvalue* expression is an *rvalue*; think of *rvalue* as "value resulting from expression"

```
double a = 0.0, b = 1.1;
double const  c = -2.0;
a = b+c; // a is lvalue; b+c is rvalue
b = std::abs(a*c); // b is lvalue; a*c is rvalue
```

Lvalues and *Rvalues*: C (4/7)

- We can use an *lvalue* when an *rvalue* is required, but we cannot use an *rvalue* when an *lvalue* is required!!!
- Useful to visualize a variable as name associated with certain memory locations

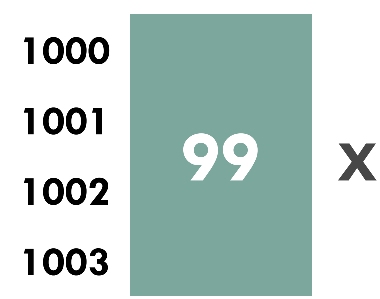
```c
int x = 99;
```

- Sometimes (as an *lvalue*) x means its memory locations and sometimes (as an *rvalue*) x means value stored in those memory locations

here, x means *lvalue*

while here, x means *rvalue*

```c
int x = 99;
x = 100;
x = x + 2;
```



Lvalues and *Rvalues*: C (5/7)

x = y

Symbol x, in this context, means "address that x represents"

This expression is termed ***lvalue***

lvalue means "x's memory location"

lvalue is known at compile-time

Symbol y, in this context, means "contents of address that y represents"

This expression is termed ***rvalue***

rvalue means "value of y"

rvalue is not known until run-time

Lvalues and *Rvalues*: C (6/7)

- Knowledge of *lvalues* and *rvalues* helps in understanding behavior of operators
  - Some operators require *lvalue* operands while others require *rvalue* operands
  - Some operators return *lvalues* while others return *rvalues*

Lvalues and *Rvalues*: C (7/7)

- Most function call expressions are *rvalues*
- Only function calls returning pointers are *lvalues*

```c
int gi = 10;

int incr(int x) { return x+1; }
int* foo()      { return &gi; }

int *pi = foo(); // ok
*pi = incr(*pi); // ok
incr(*pi) = 10;  // error
*foo() = 111;    // ok
++*foo();        // ok
```

Lvalues and *Rvalues*: C++ (1/3)

- C++ standard: *Every expression is either an lvalue or an rvalue*
- *Lvalue* expressions name objects that *persist* beyond single expression until end of scope
  - For example, a variable expression
- *Rvalue* expressions are *temporaries* that evaporate at end of full expression in which they live – at semicolon indicating sequence point
  - They are either literals or temporary objects created in the course of evaluating expressions

Lvalues and Rvalues: C++ (2/3)


```
int x = 10, *px = &x;    // x and px are lvalues
int foo(int y);          // lvalues and rvalues are expressions
std::string s{"hello"}; // s is lvalue
x = foo(20);             // ok: foo()'s return value is rvalue
px = &foo();             // error: foo()'s return value is rvalue
foo(30);                 // temporary int created for call is rvalue
int vi[10];              // vi is lvalue
vi[5] = 11;              // ok: vi[5] is lvalue
int* foobar(int *py);    // lvalues and rvalues are expressions
*foobar() = 11;          // ok: foobar()'s return value is lvalue
*px = foobar();          // ok: foobar()'s return value is lvalue
```

Lvalues and *Rvalues*: C++ (3/3)

- Another way to distinguish between *lvalueness* and *rvalueness*: Can you take address of expression?
  - Yes – expression is *lvalue*: `&x`, `&*ptr`, `&vi[5]`, …
  - No – expression is *rvalue*: `&1029`, `&(x+y)`, `&x++`, …
- Why?
  - Standard says address-of-operator requires *lvalue* operand
  - Taking address of persistent object is fine
  - But, taking address of temporary is dangerous because they evaporate quickly after expression is evaluated!!!

Increment and Decrement Operators (1/2)

- In C and C++, increment and decrement operators require *lvalue* operands
- In C, result of evaluation of all four increment and decrement expressions evaluate to *rvalue* expressions

Prefix Increment	Prefix Decrement	Postfix Increment	Postfix Decrement
++++x ✗	----x ✗	x++++ ✗	x---- ✗

Increment and Decrement Operators (2/2)

- In C and C++, increment and decrement operators require *lvalue* operands
- In C, result of evaluation of all four increment and decrement expressions evaluate to *rvalue* expressions
- In C++, pre-increment and pre-decrement operators evaluate to *lvalue* expressions

Prefix Increment	Prefix Decrement	Postfix Increment	Postfix Decrement
++++x OK	----x OK	x++++ ✕	x---- ✕

Conditional Operator ?: (1/2)

- In C, conditional operator has *higher precedence* than assignment operators and evaluates to *rvalue expression*

```
if (condition) {
  x = expression;
} else {
  y = expression;
}
```

```
int x = 1, y = 2, w = 3, z = 0;
z ? x=w : y=w;        // error
z ? (x=w) : (y=w);    // ok
```

Conditional Operator ?: (2/2)

- In C++, conditional operator has *same precedence* as assignment operators and evaluates to *lvalue expression* if both 2nd and 3rd operands are of same type and both are *lvalues*.

```
if (condition) {
  x = expression;
} else {
  y = expression;
}
```

```
condition ? x : y = expression;
```

```
int x = 1, y = 2, w = 3, z = 0;
z ? x : y = w;
```

Implicit Conversions (1/2)

- C++ allows for implicit conversions [to be compatible with C]

```
int  a = 20'000;
char c = a; // ok: squeeze large int into small char
int  b = c;
if (a != b)
  std::cout << "oops!: " << a << " != " << b << '\n';
else
  std::cout << "Wow!!! C++ has large characters\n";
```

Implicit Conversions (2/2)

- *Narrowing* conversions are unsafe!!!

```
double d = 0.;
while (std::cin >> d) {
  int i = d;
  char c = i;
  int i2 = c;
  std::cout << "d==" << d  // original double
    << " i==" << i         // converted to int
    << " i2==" << i2       // int value of char
    << " char(" << c << ")\n"; // the char
}
```

Old-Style (or, C-Style) Casts


```
double x = 10.23;
int i = (double)x; // C notation
int j = double(x); // C++ function cast notation
```

static_cast Operator


```cpp
int num = 10, den = 3;

// C++ static_cast operator converts
// one type to another related type
double result = static_cast<double>(num)/den;
```

Why Use `static_cast` Operator?

- Spotting C-style casts is difficult – an ugly cast operator name will help tools find (dangerous) casts
- C-style casts allow you to cast any type to pretty much any other type

```
int const ci = 10;
int *pi;
pi = (int*)(&ci); // ok but living dangerously!!!
pi = static_cast<int*>(&ci); // ERROR
```

Traditional C++ Initialization Syntax


```
// traditional C++ initialization syntaxes
int a = 39;
int b(39);
int c = int(39);
int d = int(); // d initialized to zero

// unsafe initialization
double e = 2.7;
int f(e);
short g = f;
```

Universal and Uniform Initialization

- C++11 introduced new initialization notation to
  - Provide universal way of initializing variables
  - Outlaw unsafe [i.e. narrowing] initializations

Universal and Uniform Initialization or List Initialization

- Modern C++ initialization syntax uses { }-list-based notation
- Compiler won't accept initializer values that will be narrowed

```
double x{}; // ok: x initialized to 0.0

int a{1'000}; // ok
char b {a};    // error: int -> char will narrow

char c{1'000}; // error: narrowing for chars
char d {48};    // ok
```

Range-for Loop

- C++ takes advantage of notion of half-open range to provide simple loop over all elements of a sequence

```
int arr[] {5, 7, 9, 4, 6, 8};
for (int element : arr) {
  std::cout << element << ' ';
}
std::cout << "\n";

int *pi{arr};
for (int element : pi) { // ERROR
  std::cout << element << ' ';
}
```

Writing To Standard Output (1/7)

Standard header declares global variables that control reading from and writing to standard streams `stdout`, `stdin`, and `stderr`

Namespaces are C++ mechanisms that introduce new scopes to avoid conflicts between names in large programs.

`std` is namespace for virtually all names in C++ standard library

```
#include <iostream>

int main() {
  std::cout << "Hello World\n";
}
```

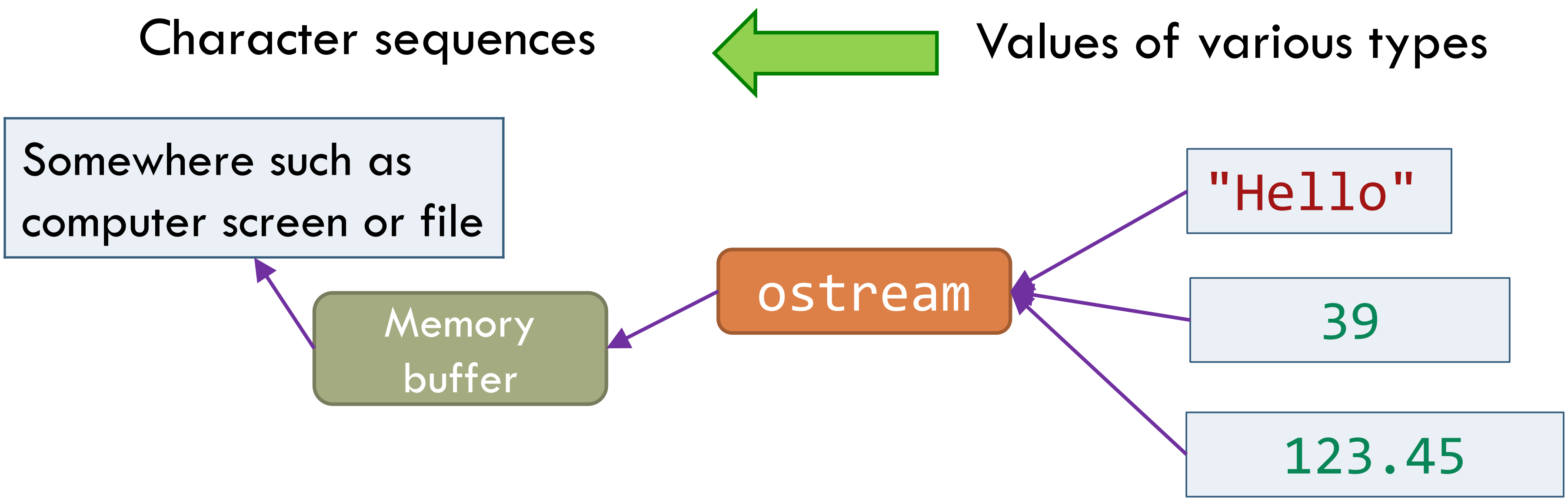
`::` is *scope resolution operator.*

`std::cout` means "name `cout` in namespace scope `std`"

Global variable of type `std::ostream` is instantiated at program startup and its purpose is to write characters to standard stream `stdout`

Writing To Standard Output (2/7)

- `std::ostream` [defined in `<ostream>`] is a type that converts objects into stream [that is, sequence] of characters [that is, bytes]
- `std::cout` is global variable of type `std::ostream` that exclusively writes to output stream `stdout`



Writing To Standard Output (3/7)

Class `std::ostream` provides member function *overloads* of binary left shift operator for built-in types [`int`, `long`, `float`, `double`, …].
Equivalent to: `(std::cout).operator<<(3+7);`

```
#include <iostream>

int main() {
  std::cout << 3+7;
  std::cout << "Hello World\n";
}
```

Class `std::ostream` provides non-member function *overloads* of binary left shift operator for inserting characters [`char`, `unsigned char`, `char const*`, …].
Equivalent to: `std::operator<<(std::cout, "Hello World\n");`

Writing To Standard Output (4/7)

Expression equivalent to: `(std::cout).operator<<(3+7)`

Expression evaluates to: `std::cout`

Binary left shift operator `<<` has *lower precedence* than operator `+` and is *left-associative*

```
#include <iostream>

int main() {
  std::cout << 3+7 << "\n";
}
```

Expression equivalent to: `std::cout << "\n"`

Expression evaluates to: `std::operator<<(std::cout, "\n")`

Writing To Standard Output (5/7)


```
#include <iostream>

int main() {
  std::cout << 3+7 << "\n" ;
}
```

1) Expression equivalent to function call:

`std::operator<<( (std::cout).operator<<(3+7), "\n" )`

2) 1st argument is *member function* call by variable `std::cout` which writes characters equivalent to integral value `10` to standard output stream and returns [a reference to] `std::cout`

3) What remains to be evaluated is call to function:

`std::operator<<( std::cout, "\n" )`

which writes newline to standard output stream and return value from the function call is [a reference to] `std::cout`

Writing To Standard Output (6/7)


```
#include <iostream>

int main() {
  std::cout << "Hello World\n";
  std::cout << "Hello World" << std::endl;
}
```

`std::endl` is output manipulator.

Manipulators are helper functions that change the way a stream formats characters.

Here `std::endl` does two things to `stdout` – the stream to which `std::cout` is writing characters:

1) Outputs newline character `'\n'`
2) Flushes output stream `stdout`

Writing To Standard Output (7/7)


```cpp
#include <iostream>

int main() {
  std::cout << "Hello World\n";
  std::cout << 3+7 << "\n";
  std::cout << 3.1*7.2 << std::endl;
}
```

```cpp
( (std::cout).operator<<(22.32) ).operator<<(std::endl);
```

Reading From Standard Input (1/5)

Global variable of type std::istream instantiated at program startup to write characters to standard stream stdin

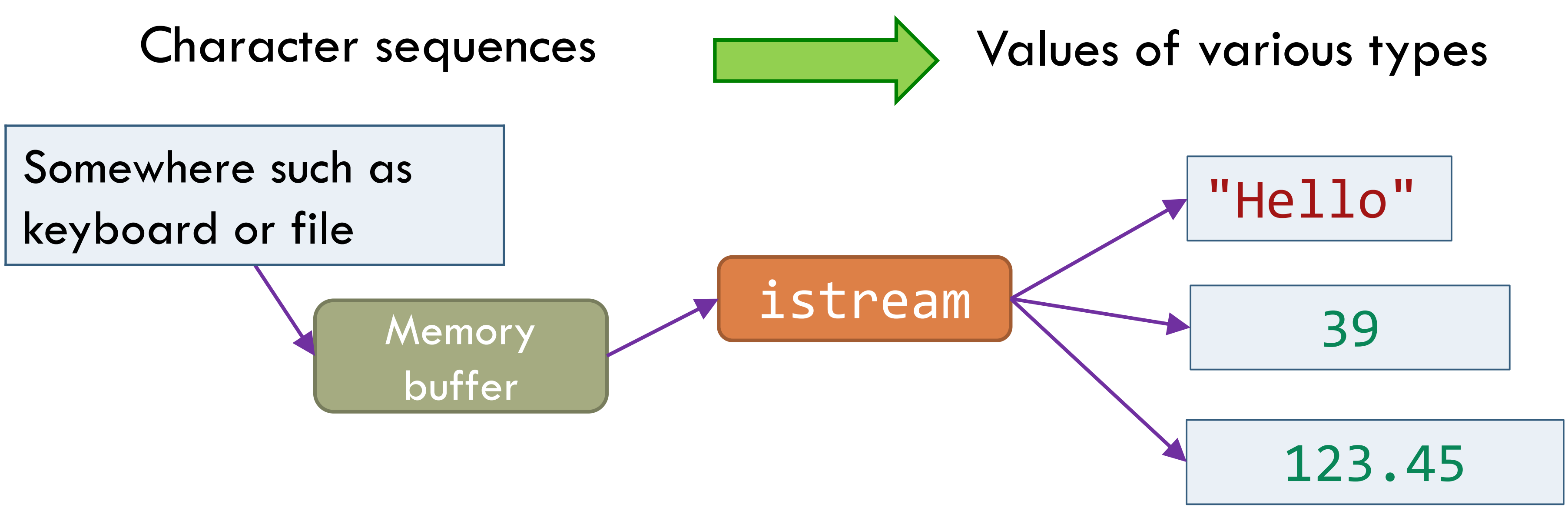
```
#include <iostream>

int main() {
  std::cout << "Enter your first name: ";
  char name[81];
  std::cin >> name;
  std::cout << "Hello " << name << '\n';
}
```

Class std::istream provides non-member function *overloads* of binary right shift operator for extracting characters [char, unsigned char, char const*, …] that is equivalent to:

```
std::operator>>(std::cin, name);
```

Reading From Standard Input (2/5)

- `std::istream` [defined in `<istream>`] is a type that converts stream [that is, sequence] of characters [that is, bytes] to typed objects
- `std::cin` is global variable of type `std::istream` that exclusively reads from input stream `stdin`



Reading From Standard Input (3/5)


```
#include <iostream>

int main() {
  std::cout << "Enter your first name and age: ";
  char name[81];
  std::cin >> name;
  int age;
  std::cin >> age;
  std::cout << "Hello " << name << " age " << age << "\n";
}
```

Class `std::istream` provides member function *overloads* of binary right shift operator for built-in types [`int`, `long`, `float`, `double`, …] that is equivalent to: `(std::cin).operator>>(age);`

Reading From Standard Input (4/5)

Expression equivalent to: `std::operator>>(std::cin, name)`
Expression evaluates to: `std::cin`

```
#include <iostream>

int main() {
  std::cout << "Enter your first name and age: ";
  char name[81];
  int age;
  std::cin >> name >> age;
  std::cout << "Hello " << name << " age " << age << "\n";
}
```

Binary right shift operator >> is *left-associative*

Expression equivalent to: `(std::cin).operator>>(age)`
Expression evaluates to: `std::cin`

Reading From Standard Input (5/5)


```
#include <iostream>

int main() {
  std::cout << "Enter your first name and age: ";
  char name[81];
  int age;
  std::cin >> name >> age;
  std::cout << "Hello " << name << " age " << age << "\n";
}
```

Expression equivalent to:

```
std::operator>>(std::cin, name).operator>>(age)
```

Ensuring Validity of Input Data

- Surprisingly hard to write robust programs that can survive incorrect or wrong data entered by untrained or careless users!!!

`std::istream` States (1/2)

- What happens when following program is executed?

```cpp
// input-error01.cpp
std::cout << "Enter integer value: ";
int i;
std::cin >> i; // Enter .75

std::cout << "Enter fractional value: ";
double d = 1.1;
std::cin >> d; // Enter 123.45
std::cout << "i==" << i << " d==" << d << '\n';
```

Since 1st character encountered by `cin` is `'.'` [which cannot be part of any `int` value], `cin` enters a *failed* state!!!
Therefore, subsequent expression `std::cin >> d` has no effect!!!

`std::istream` States (2/2)

- `std::istream` can be in one of four states:

Stream states	
`good()`	Operations succeeded
`eof()`	We hit end of input [that is, end of file]
`fail()`	Something unexpected happened [e.g., looking for a digit character and found `'.'`]
`bad()`	Something unexpected and serious happened [e.g., a disk read error]

```
std::cout << "Enter integer value: ";
int i;
std::cin >> i; // Enter .75
if (std::cin.fail()) {
  std::cout << "Bad input.\n";
  // Exit program? How to get correct input?
}
```

Input: Error Handling (1/3)

- Both `istream` and `ostream` inherit a function to clear stream's error state and retry read or write operation ...

```
std::cout << "Enter integer value: ";
int i;
std::cin >> i; // user enters .75
if (std::cin.fail()) {
  std::cin.clear(); // clear error state
  std::cin >> i;    // retry again ...
}
std::cout << "i==" << i << "\n";
```

Retry won't work after clearing input stream's error state since characters `'.'`, `'7'`, `'5'` are still in input stream!!!

We must tell `std::cin` to ignore these characters using inherited `ignore()` function

Input: Error Handling (2/3)


```
std::cout << "Enter integer value: ";
int i;
std::cin >> i; // user enters .75
if (std::cin.fail()) {
  std::cin.clear(); // clear error state
  std::cin.ignore(1000, '\n');
  std::cin >> i;    // retry again ...
}
std::cout << "i==" << i << "\n";
```

`std::cin` will ignore either first `1000` characters in stream or all character up to and including delimiting character `'\n'` – whichever occurs first

Input: Error Handling (3/3)


```cpp
// input-error02.cpp: more robust version ...
int i;
// as long as input stream cannot read an integer value,
// continue prompting user to provide integer value ...
do {
  if (std::cin.fail()) {
    std::cin.clear(); // clear stream's error state
    std::cin.ignore(1000, '\n'); // ignore characters ...
  }
  std::cout << "Enter integer value: ";
  std::cin >> i;
} while (!std::cin.good());

// ok: finally, we've valid integer data ...
std::cout << "i==" << i << "\n";
```

Anything Wrong Here?

- What happens when following program is executed?

```
// input-error03.cpp:
int main() {
  char str[10];
  int i;
  std::cin >> str >> i;
  // user enters: Supercalifragilistic
  std::cout << "You entered: " << str << " | " << i << "\n";
}
```

Since `sizeof("Supercalifragilistic") > sizeof(str)`, characters typed by user will overflow static array `str` causing stack to be smashed!!!

One solution is to limit program to read only first 9 characters …

Setting Field Width for Input (1/2)

- Both istream and ostream inherit a function to manage maximum number of characters read from or written to stream

```
// incorrect version
char str[10];
std::streamsize old_width = std::cin.width(sizeof(str));
std::cin >> str; // user enters: Supercalifragilistic
std::cin.width(old_width);

int i;
std::cin >> i;
std::cout << "You entered: " << str << " | " << i << "\n";
```

Makes cin read maximum of 9 characters

Reads first 9 characters Supercali but remaining characters fragilistic will be read in expression std::cin >> i and will cause stream to be in fail state!!! We must tell std::cin to ignore characters fragilistic using inherited ignore() function.

Setting Field Width for Input (2/2)


```cpp
// input-error04.cpp: more robust version ...
char str[10];
// read maximum 9 characters ...
std::streamsize old_width = std::cin.width(sizeof(str));
std::cin >> str; // user enters: Supercalifragilistic

std::cin.width(old_width);
// ignore any other characters in stream including '\n'
std::cin.ignore(1000, '\n');

int i;
std::cin >> i;
std::cout << "You entered: " << str << " | " << i << "\n";
```

Idiomatic Testing of Streams

- `ostream` or `istream` variable can be used as condition by calling this member function
- In that case, condition is `true` (succeeds) or `false` (fails) if stream's state is `good()` or `!good()`, respectively

```
int i;
do {
  if (std::cin.fail()) {
    std::cin.clear(); // clear stream's error state
    std::cin.ignore(1000, '\n'); // ignore characters ...
  }
  std::cout << "Enter integer value: ";
  std::cin >> i;
} while (!std::cin);

// ok: finally, we've valid integer data ...
std::cout << "i==" << i << "\n";
```

means "`while std::cin` is not good"

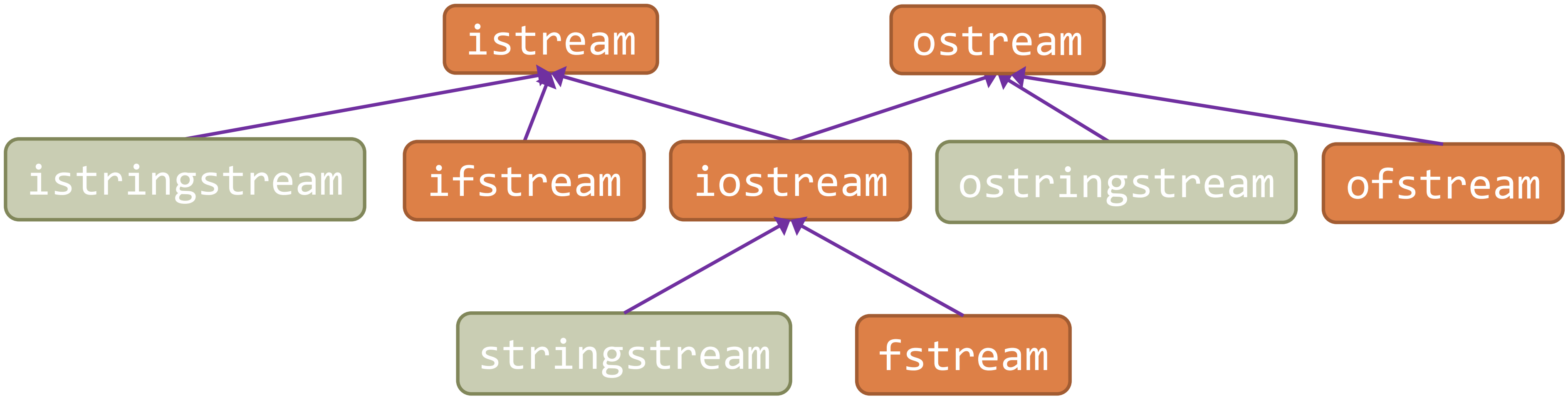
Idiomatic C++ Input Loops

- Example that detects adjacent repeated positive `int` values in sequence of values

```
int previous = -1, current;
// read unknown number of +ve integer values from stdin ...
// signal EOF in Linux using CTRL-D ...
while (std::cin >> current) {
  if (current < 0) continue;
  if (previous == current) {
    std::cout << "repeated value: " << current << '\n';
  }
  previous = current;
}
```

I/O Streams Hierarchy

- `std::istream` can be connected to input device [e.g., keyboard], file, or `std::string` [a C++ standard library type]
- `std::ostream` can be connected to output device [console window], file, or `std::string`



File Streams (1/2)

- File stream can be opened either by constructor or by an `open()` call:

Opening files with file stream	
`std::fstream fs;`	Make a file stream variable for opening later
`fs.open(s, m);`	Open a file called `s` [C-style string] with mode `m` and have variable [defined in previous row] `fs` refer to it
`std::fstream fs(s, m);`	Open a file called `s` [C-style string] with mode `m` and make a file stream `fs` refer to it
`fs.is_open();`	Is file referenced by file stream `fs` open?
`fs.close();`	Close file referenced by file stream `fs`

File Streams (2/2)

- You can open a file in one of several modes:

Opening files with file stream	
`std::ios_base::in`	Open file for reading
`std::ios_base::out`	Open file for writing
`std::ios_base::app`	Open file for appending [i.e., add from end of the file]
`std::ios_base::binary`	Open file so that operations are performed in binary [as opposed to text]
`std::ios_base::ate`	"at end [of file]" [open and seek to the end]
`std::ios_base::trunc`	Truncate file to zero length

File I/O

- File for reading is attached to `istream`
- File for writing is attached to `ostream`
- Since we know how to read from `istream` and write to `ostream`, anything you could do to output stream `stdout` and input stream `stdin`, you can do to files too …
- See *fileio.cpp*

Default Formatting of Output


```
#include <iostream>

int main() {
  for (int i=1; i < 200'000'000; i *= 10) {
    std::cout << "Number: " << i << '\n';
  }
}
```

Program output:

```
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Stream Manipulators (1/2)

- C++ standard library provides various helper functions to modify default formatting
  - Setting base of integer values
  - Setting field widths
  - Setting precision of floating-point values
  - Setting fill character in output stream
  - Setting alignment
  - Skipping whitespace in input stream
  - Inserting newline and flushing output stream
  - …

Setting Character Width of Value to Write (1/4)

- Non-sticky manipulator `std::setw` used to specify minimum number of characters to be printed for next value
- By default, values are written with right alignment

```
#include <iostream>
#include <iomanip>

int main() {
  std::cout << "Width is 6.\n";
  std::cout << '+' << std::setw(6)
            << 1 << 2 << 3 << '\n';
}
```

applies only to immediate value written to standard output

Program output:

```
Width is set to 6
+     123
```

Setting Character Width of Value to Write (2/4)


```cpp
#include <iostream>
#include <iomanip>

int main() {
  std::cout << "Width is 6.\n";
  std::cout << '+' << std::setw(6)
            << 1 << std::setw(6) << 2 << 3 << '\n';
}
```

only applies to immediate value written to standard output

Program output:

```
Width is set to 6
+     1     23
```

Setting Character Width of Value to Write (3/4)


```
std::cout << "Width is 6.\n";
for (int i=1; i < 200'000'000; i *= 10) {
  std::cout << "Number: " << std::setw(6) << i << '\n';
}
std::cout << "Width is 6.\n";
std::cout << '+' << std::setw(6) << 1 << 2 << 3 << '\n';
```

Program output:

```
Width is 6.
Number:      1
Number:     10
Number:    100
Number:   1000
Number:  10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
Width is 6.
+     123
```

Setting Character Width of Value to Write (4/4)


```
std::cout << "Width is 9.\n";
for (int i=1; i < 200'000'000; i *= 10) {
  std::cout << "Number: " << std::setw(9) << i << '\n';
}
std::cout << "Width is 6.\n";
std::cout << '+' << std::setw(6) << 1 << 2 << 3 << '\n';
```

Program output:

```
Width is 9.
Number:         1
Number:        10
Number:       100
Number:      1000
Number:     10000
Number:    100000
Number:   1000000
Number:  10000000
Number: 100000000
Width is 6.
+     123
```

Padding Character

- Default padding character is space `' '`
- Specify padding character with `setfill` manipulator

```
for (int i=1; i < 200'000'000; i *= 10)
  std::cout << "Number: " << std::setfill('.')
            << std::setw(9) << i << '\n';
```

Program output:

```
Number: ........1
Number: .......10
Number: ......100
Number: .....1000
Number: ....10000
Number: ...100000
Number: ..1000000
Number: .10000000
Number: 100000000
```

Left Alignment

- Default is right alignment
- Specify left alignment using `left` manipulator

```
for (int i=1; i < 200'000'000; i *= 10)
  std::cout << "Number: " << std::setfill('.')
    << std::left << std::setw(9) << i << '\n';
```

Program output:

```
Number: 1........
Number: 10.......
Number: 100......
Number: 1000.....
Number: 10000....
Number: 100000...
Number: 1000000..
Number: 10000000.
Number: 100000000
```

Changing Base of Integral Values (1/2)

- Numbers are by default printed in decimal

```
for (int i{}; i < 16; ++i) {
  for (int j{}; j < 16; ++j)
    cout << i*16+j << ' ';
  cout << '\n';
}
```

Program output:

```
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 99 100 101 102 103 104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
```

Changing Base of Integral Values (2/2)

- Specify base for integral value using *sticky* manipulator `setbase` [argument is 8 for octal & 16 for hex values]

```
cout << setbase(16);
for (int i{}; i < 16; ++i) {
  for (int j{}; j < 16; ++j)
    cout << i*16+j << ' ';
  cout << '\n';
}
```

Program output:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

Printing Base of Integral Values

- Base for integral value can be printed to output stream using *sticky* manipulator `showbase`

```
cout << setbase(16);
cout << showbase;
for (int i{}; i < 16; ++i) {
  for (int j{}; j < 16; ++j)
    cout << i*16+j << ' ';
  cout << '\n';
}
```

Program output:

```
0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa 0xb 0xc 0xd 0xe 0xf
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e 0x1f
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 0x28 0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f
0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x3a 0x3b 0x3c 0x3d 0x3e 0x3f
0x40 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48 0x49 0x4a 0x4b 0x4c 0x4d 0x4e 0x4f
0x50 0x51 0x52 0x53 0x54 0x55 0x56 0x57 0x58 0x59 0x5a 0x5b 0x5c 0x5d 0x5e 0x5f
0x60 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68 0x69 0x6a 0x6b 0x6c 0x6d 0x6e 0x6f
0x70 0x71 0x72 0x73 0x74 0x75 0x76 0x77 0x78 0x79 0x7a 0x7b 0x7c 0x7d 0x7e 0x7f
0x80 0x81 0x82 0x83 0x84 0x85 0x86 0x87 0x88 0x89 0x8a 0x8b 0x8c 0x8d 0x8e 0x8f
0x90 0x91 0x92 0x93 0x94 0x95 0x96 0x97 0x98 0x99 0x9a 0x9b 0x9c 0x9d 0x9e 0x9f
0xa0 0xa1 0xa2 0xa3 0xa4 0xa5 0xa6 0xa7 0xa8 0xa9 0xaa 0xab 0xac 0xad 0xae 0xaf
0xb0 0xb1 0xb2 0xb3 0xb4 0xb5 0xb6 0xb7 0xb8 0xb9 0xba 0xbb 0xbc 0xbd 0xbe 0xbf
0xc0 0xc1 0xc2 0xc3 0xc4 0xc5 0xc6 0xc7 0xc8 0xc9 0xca 0xcb 0xcc 0xcd 0xce 0xcf
0xd0 0xd1 0xd2 0xd3 0xd4 0xd5 0xd6 0xd7 0xd8 0xd9 0xda 0xdb 0xdc 0xdd 0xde 0xdf
0xe0 0xe1 0xe2 0xe3 0xe4 0xe5 0xe6 0xe7 0xe8 0xe9 0xea 0xeb 0xec 0xed 0xee 0xef
0xf0 0xf1 0xf2 0xf3 0xf4 0xf5 0xf6 0xf7 0xf8 0xf9 0xfa 0xfb 0xfc 0xfd 0xfe 0xff
```

Exercise

- Get pretty output that looks like this:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

Default Format of Floating-Point Values

- Whether floating point value is printed using fixed or scientific format is determined by its value

```
long double pi = 22.0/7.0;
for (size_t i{1}; i <= 1'000'000'000; i*=10)
  std::cout << (pi/static_cast<long double>(i)) << '\n';
```

Program output:

```
3.14286
0.314286
0.0314286
0.00314286
0.000314286
3.14286e-05
3.14286e-06
3.14286e-07
3.14286e-08
3.14286e-09
```

Fixed Point Format

- Specify specific format such as fixed with *sticky* `fixed` manipulator declared in `<ios>`

```
long double pi = 22.0/7.0;
std::cout << std::fixed;
for (size_t i{1}; i <= 1'000'000'000; i*=10)
  std::cout << (pi/static_cast<long double>(i)) << '\n';
```

Program output:

```
3.142857
0.314286
0.031429
0.003143
0.000314
0.000031
0.000003
0.000000
0.000000
0.000000
```

Scientific Format

- Specify specific format as scientific with `scientific` manipulator declared in `<ios>`

```
long double pi = 22.0/7.0;
std::cout << std::scientific;
for (size_t i{1}; i <= 1'000'000'000; i*=10)
  std::cout << (pi/static_cast<long double>(i)) << '\n';
```

Program output:

```
3.142857e+00
3.142857e-01
3.142857e-02
3.142857e-03
3.142857e-04
3.142857e-05
3.142857e-06
3.142857e-07
3.142857e-08
3.142857e-09
```

Precision (1/3)

- Number of digits to right of decimal point is controlled by sticky manipulator `setprecision` or sticky `ios_base` member function `precision`

```
3.142857142857
0.314285714286
0.031428571429
0.003142857143
0.000314285714
0.000031428571
0.000003142857
0.000000314286
0.000000031429
0.000000003143
```

```cpp
long double pi = 22.0/7.0;
std::cout << std::fixed;
for (size_t i{1}; i <= 1'000'000'000; i*=10)
  std::cout << std::setprecision(12) <<
   (pi/static_cast<long double>(i)) << '\n';
```

Program output:

Precision (2/3)

- Combine scientific format with precision

```
long double pi = 22.0/7.0;
std::cout << std::scientific;
std::cout.precision(12);
for (size_t i{1}; i <= 1'000'000'000; i*=10)
  std::cout << (pi/static_cast<long double>(i))
            << '\n';
```

Program output:

```
3.1428571429e+00
3.1428571429e-01
3.1428571429e-02
3.1428571429e-03
3.1428571429e-04
3.1428571429e-05
3.1428571429e-06
3.1428571429e-07
3.1428571429e-08
3.1428571429e-09
```

Precision (3/3)

- Calling `ios_base` member function `precision` member function with no argument returns current precision setting
- Good practice to save current precision setting and restore it later …

```
long double pi = 22.0/7.0;
std::streamsize old_sz = std::cout.precision();
std::cout << std::scientific << std::setprecision(12);
for (size_t i{1}; i <= 1'000'000'000; i*=10)
  std::cout << (pi/static_cast<long double>(i)) << '\n';
std::cout.precision(old_sz);
```

Exercise

- Print floating-point values aligned on decimal point [as shown below]. Assume four spaces each for both integer and fractional parts.

```
   0.5000
   1.3450
  23.7890
 456.1234
1456.1234
```