MODULE 1 / UNIT 0

BASIC PROGRAMMING SKILLS

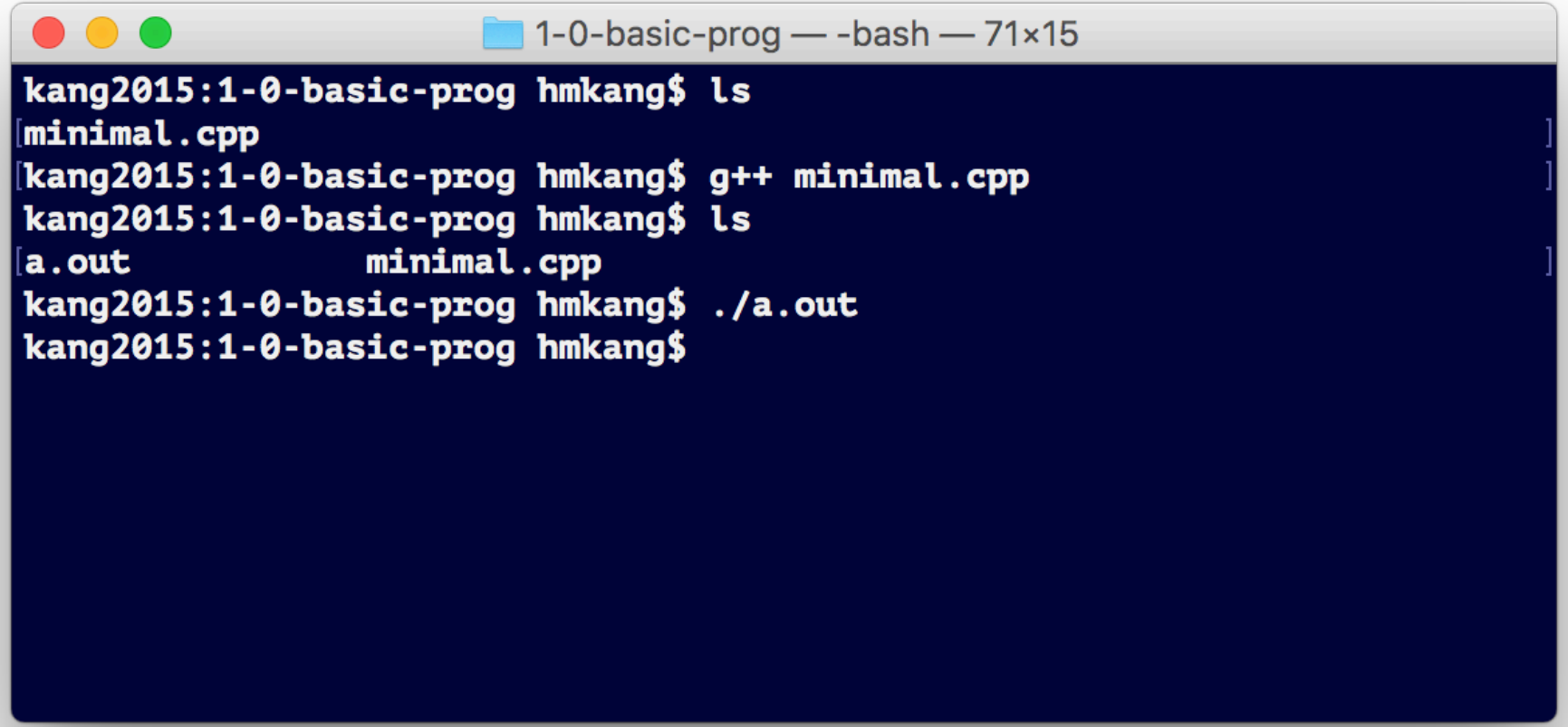
Today

- Compiler vs Interpreter programming languages
- Fundamental data types in C++
- Input and output in C++
- Command line arguments
- Python vs R vs C++ programming
- Control structure

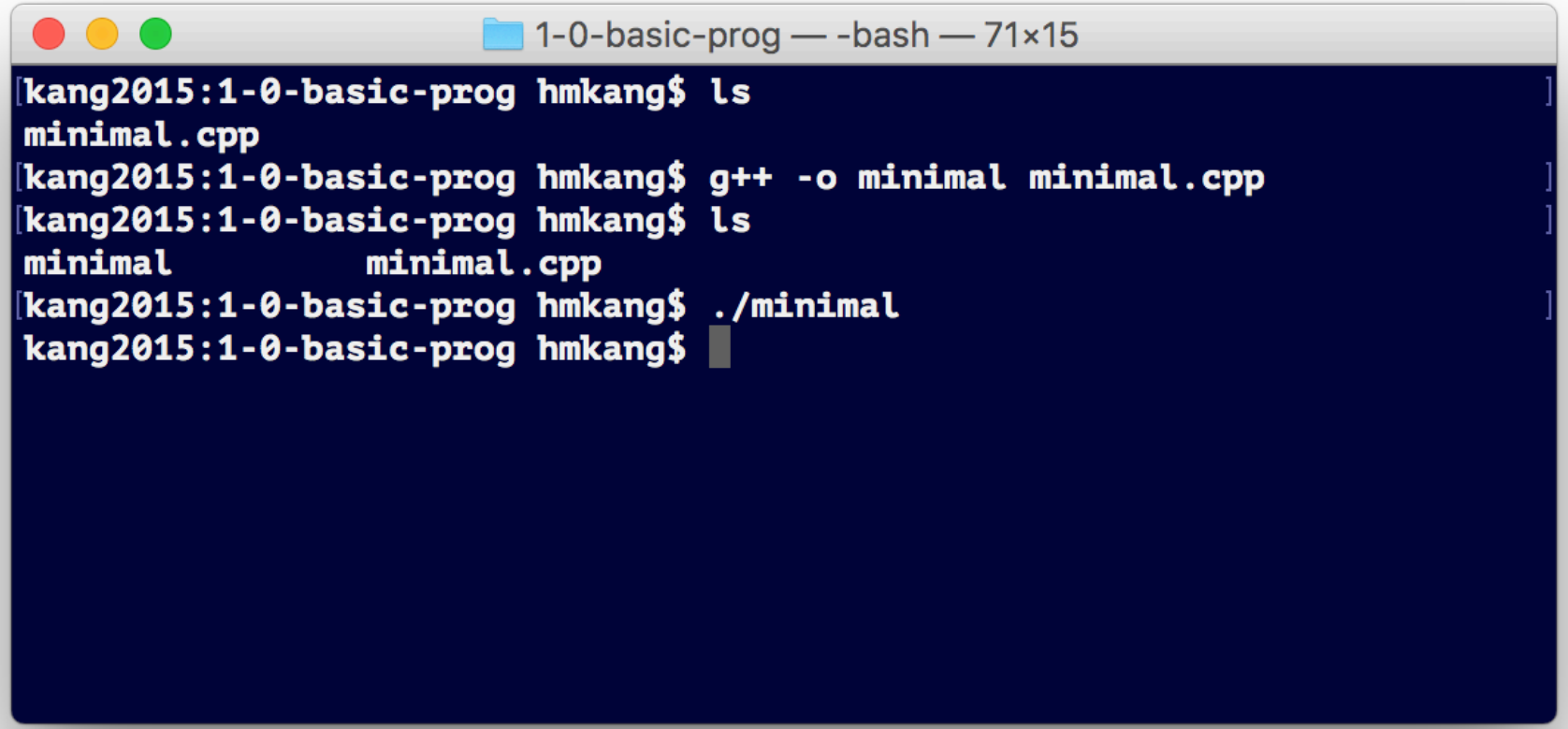
A minimal C++ program

```
1  int main(int argc, char** argv) {
2    return 0;
3  }
4
```

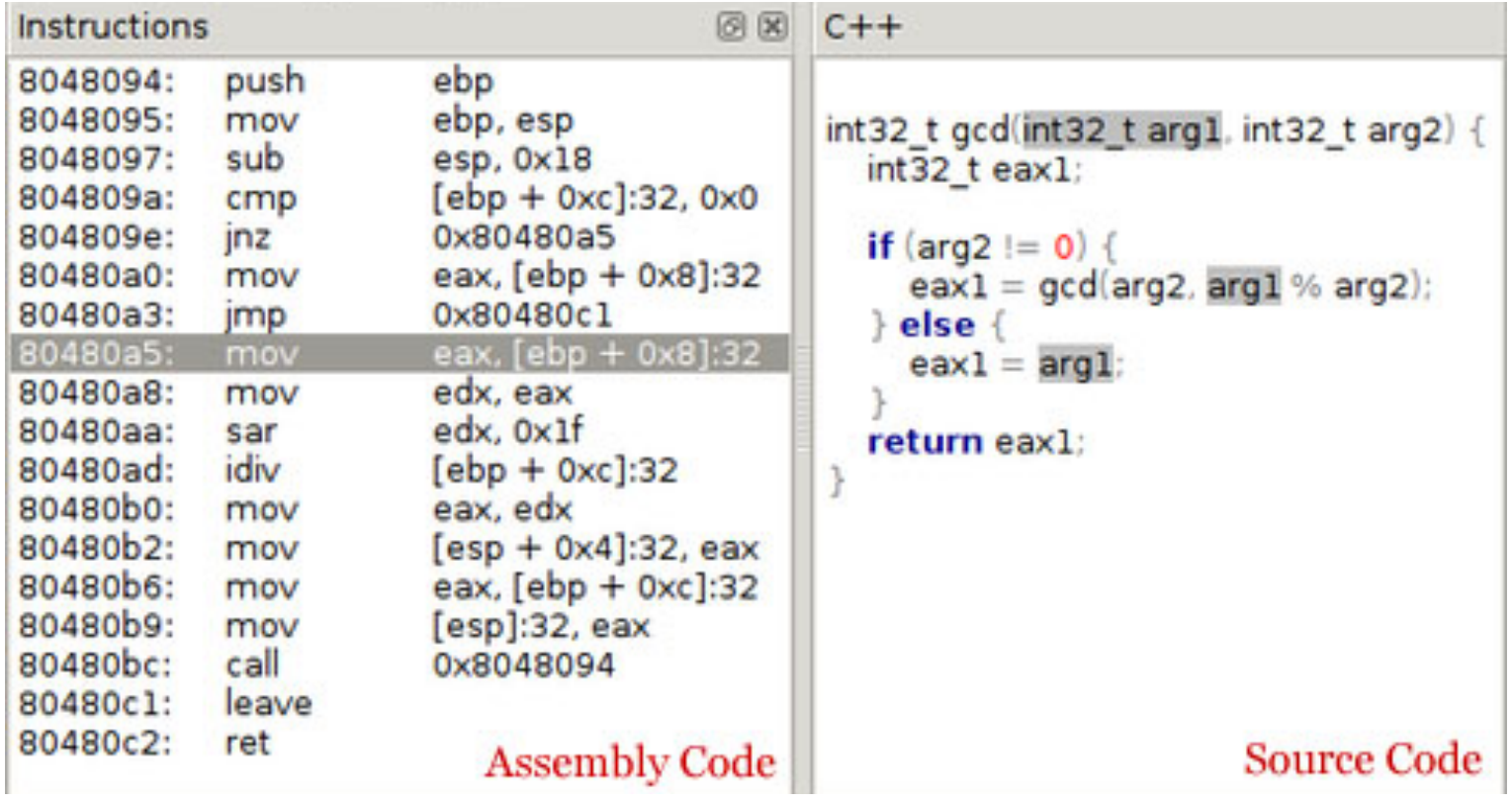
To run, compile first.



Specifying the binary program name..

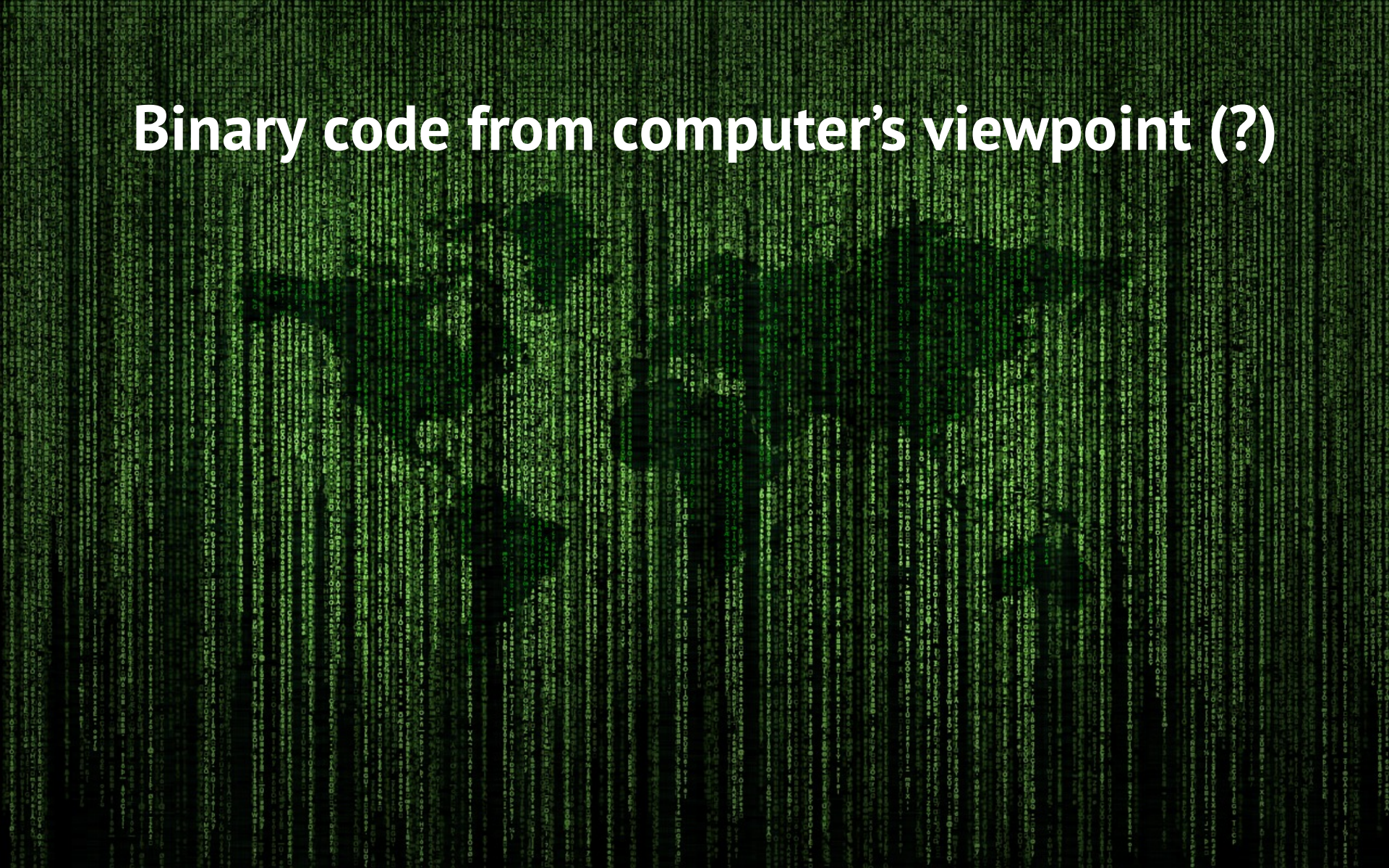


How does a compiler work?



http://techwelkin.com/compiler-vs-interpreter

Binary code from computer's viewpoint (?)



Compiler vs Interpreter language = Batch vs Simultaneous Translation

나는 동시통역하는 Edit naneun dongsitong-yeoghaneun	I am a simultaneous translator

Compiler vs Interpreter language = Batch vs Simultaneous Translation

나는 동시통역하는 Edit naneun dongsitong-yeoghaneun	I am a simultaneous translator

나는 동시통역하는 것이 더 유용하기 Edit naneun dongsitong-yeoghaneun geos-i	I think it is more useful to have simultaneous interpretation

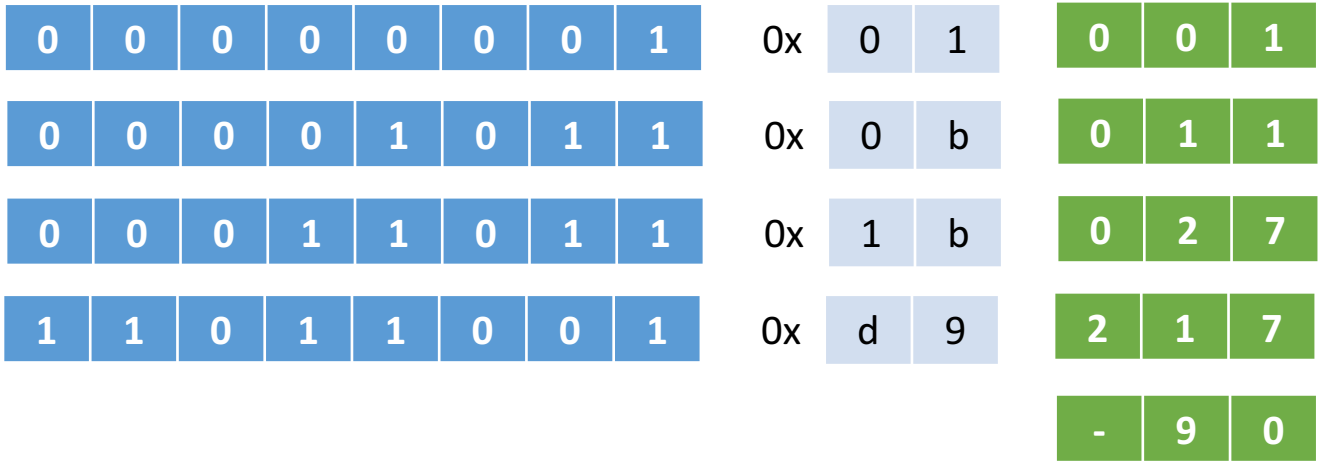
Compiler : Interpreter =
Batch : Simultaneous translation

나는 동시통역하는 Edit naneun dongsitong-yeoghaneun	I am a simultaneous translator
나는 동시통역하는 것이 더 유용하기 Edit naneun dongsitong-yeoghaneun geos-i	I think it is more useful to have simultaneous interpretation
나는 동시통역하는 것이 더 유용하기 때문에 좋아한다. Edit	I like it because simultaneous interpretation is more useful.

Understanding binary views

- **Bit : A single binary digit** 0 1

- **Byte : 8 bits**



What is the range of integers that a byte can represent?

Fundamental integer data types in C++

Type in <stdint.h>	Standard Type	Range
int8_t	signed char	-128 – 127
uint8_t	(unsigned) char	0 – 255
int16_t	(signed) short	-32,768 – 32,767
uint16_t	unsigned short	0 – 65,536
int32_t	(signed) long/int	-2,147,483,648 – 2,147,483,647
uint32_t	unsigned long/int	0 – 4,294,967,296
int64_t	(signed) long long	-9,223,372,036,854,775,808 – 9,223,372,036,854,775,807
uint64_t	unsigned long long	– 18,446,744,073,709,551,616

Most common settings in modern computers, but details may vary by platform.

Fundamental floating-point data types in C++

Standard Type	Bytes	# Significant Decimal Digits	Range
float (single precision)	4	7 (6 – 9)	$\pm 3 \times 10^{-38} - 3 \times 10^{38}$
double (double precision)	8	15 (15 – 17)	$\pm 2 \times 10^{-308} - 2 \times 10^{308}$
long double (quadruple precision)	16	34 (33 – 36)	$\pm 1 \times 10^{-4932} - 1 \times 10^{4932}$

Most common settings in modern computers, but details may vary by platform.

What is the difference between `c` and `f`?

```
float a = 1.0e9, b = 1.0;
float c = a - b;

double d = 1.0e9, e = 1.0;
double f = d - e;
```

and how can we verify the answer?

Understanding <iostream>

- **Input/output is the means that computers communicate with users via devices, including but not limited to:**
  - Displaying messages to users on the screen
  - Receiving messages from users via keyboard
  - Sending/receiving messages by reading/writing files.

- **Standard I/O interfaces:**
  - standard input : default is keyboard input
  - standard output : default is printing to screen
  - standard error: default is printing to screen

Key elements in `<iostream>`

Need to include "#include <iostream>" before using these variables.

- **`std::cin` – standard input (keyboard input by default)**
- **`std::cout` – standard output (screen printout by default)**
  - **`std::cout.precision(int)`** – set # of significant decimal digits
- **`std::cerr` – standard error (screen printout by default)**
- **`std::endl` – a constant representing a new line.**
- **`>>` : an operator used to send messages to `std::cout`**
- **`<<` : an operator used to receive messages from `std::cin`**

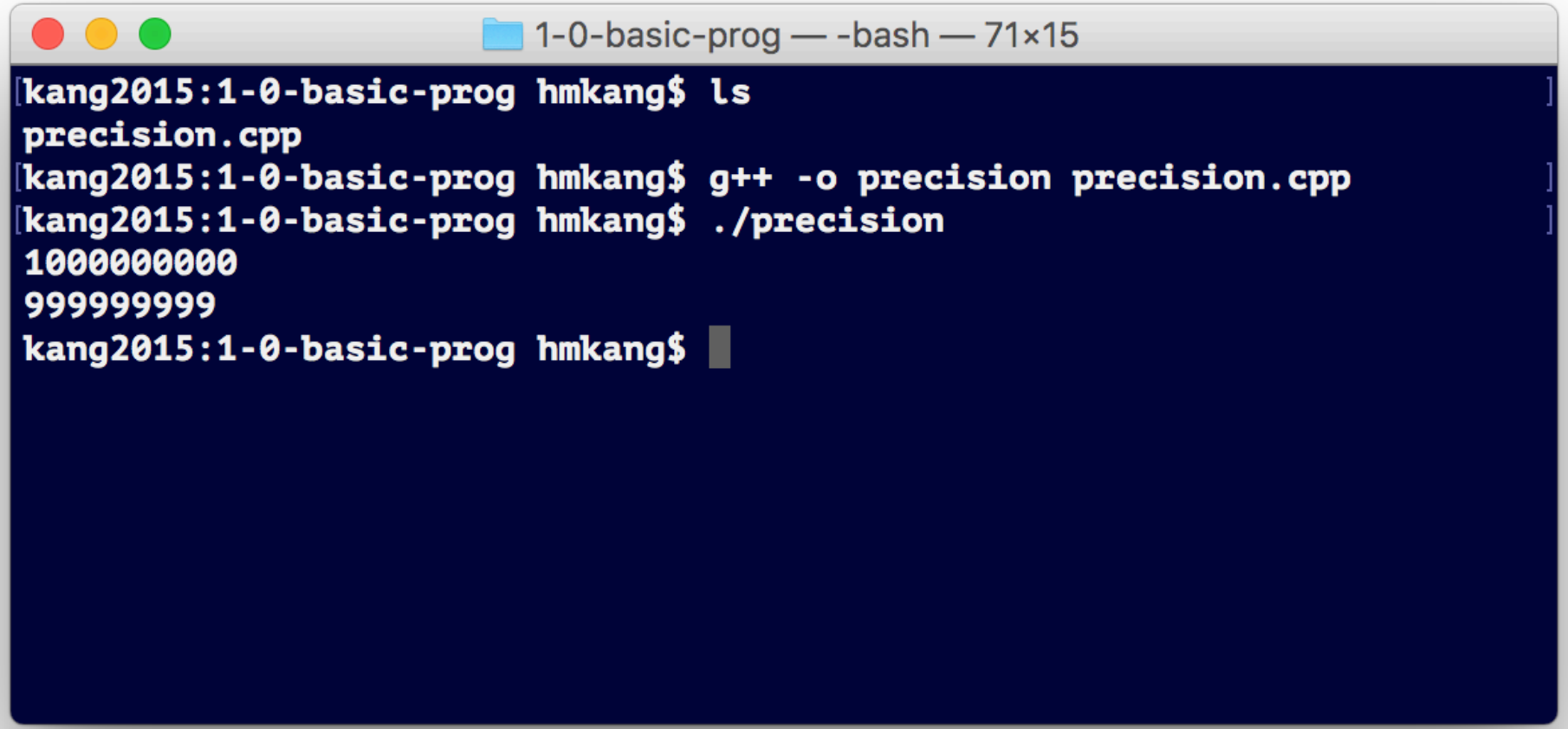
First C++ code that actually does something

```cpp
#include <iostream>

int main(int argc, char** argv) {
  float a = 1.0e9, b = 1.0;
  float c = a - b;

  double d = 1.0e9, e = 1.0;
  double f = d - e;

  std::cout.precision(10);
  std::cout << c << std::endl;
  std::cout << f << std::endl;

  return 0;
}
```

What do you expect to see?

Why?

An example output



If you don't want to keep typing `std::`...

```cpp
#include <iostream>
using namespace std;
int main(int argc, char** argv) {
  float a = 1.0e9, b = 1.0;
  float c = a - b;

  double d = 1.0e9, e = 1.0;
  double f = d - e;

  cout.precision(10);
  cout << c << endl;
  cout << f << endl;

  return 0;
}
```

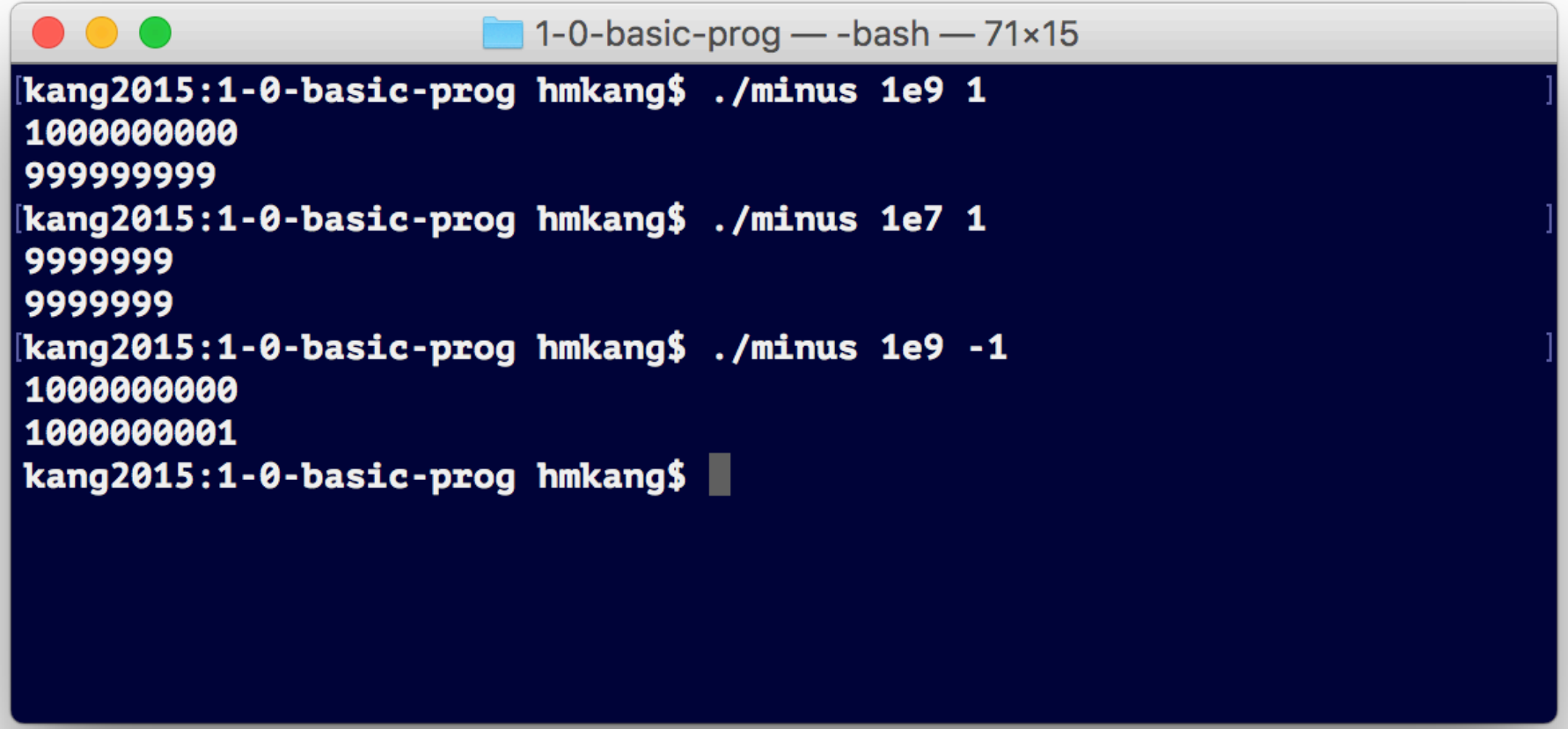
This will do the exactly the same thing as before

If you don't want to keep typing `std::`...

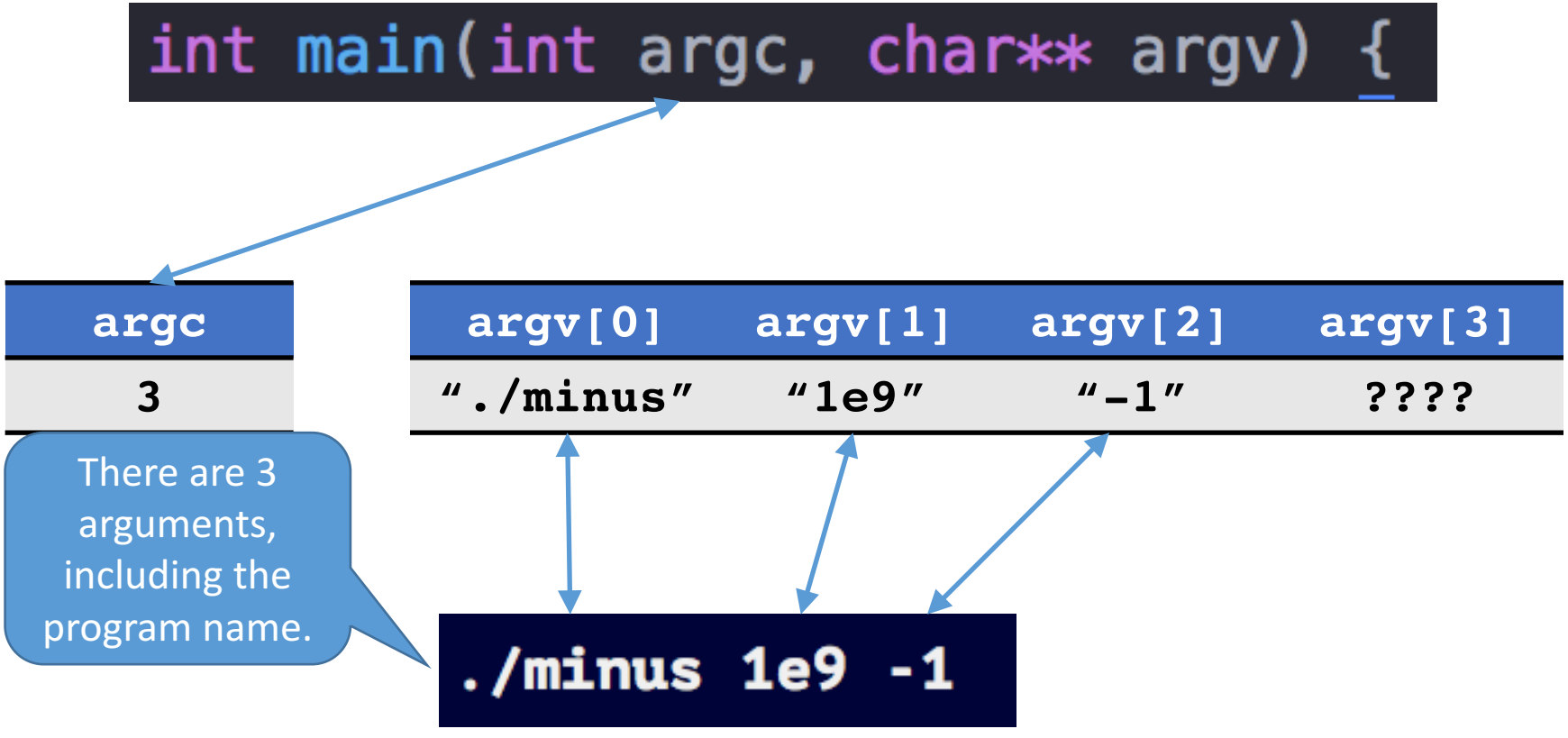
```
1  #include <iostream>
2  using namespace std;
3  int main(int argc, char** argv) {
4    float a = 1.0e9, b = 1.0;
5    float c = a - b;
6
7    double d = 1.0e9, e = 1.0;
8    double f = d - e;
9
10   cout.precision(10);
11   cout << c << endl;
12   cout << f << endl;
13
14   return 0;
15 }
```

This will do the exactly the same thing as before

Can we make it flexible? For example…



Interfacing with command-line arguments



Converting `char*` to `float` or `double`

- C-style way to represent a string:
  - `char* s = "Hello";`

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]
'H'	'e'	'l'	'l'	'o'	'\0'

- Converting string to numeric numbers

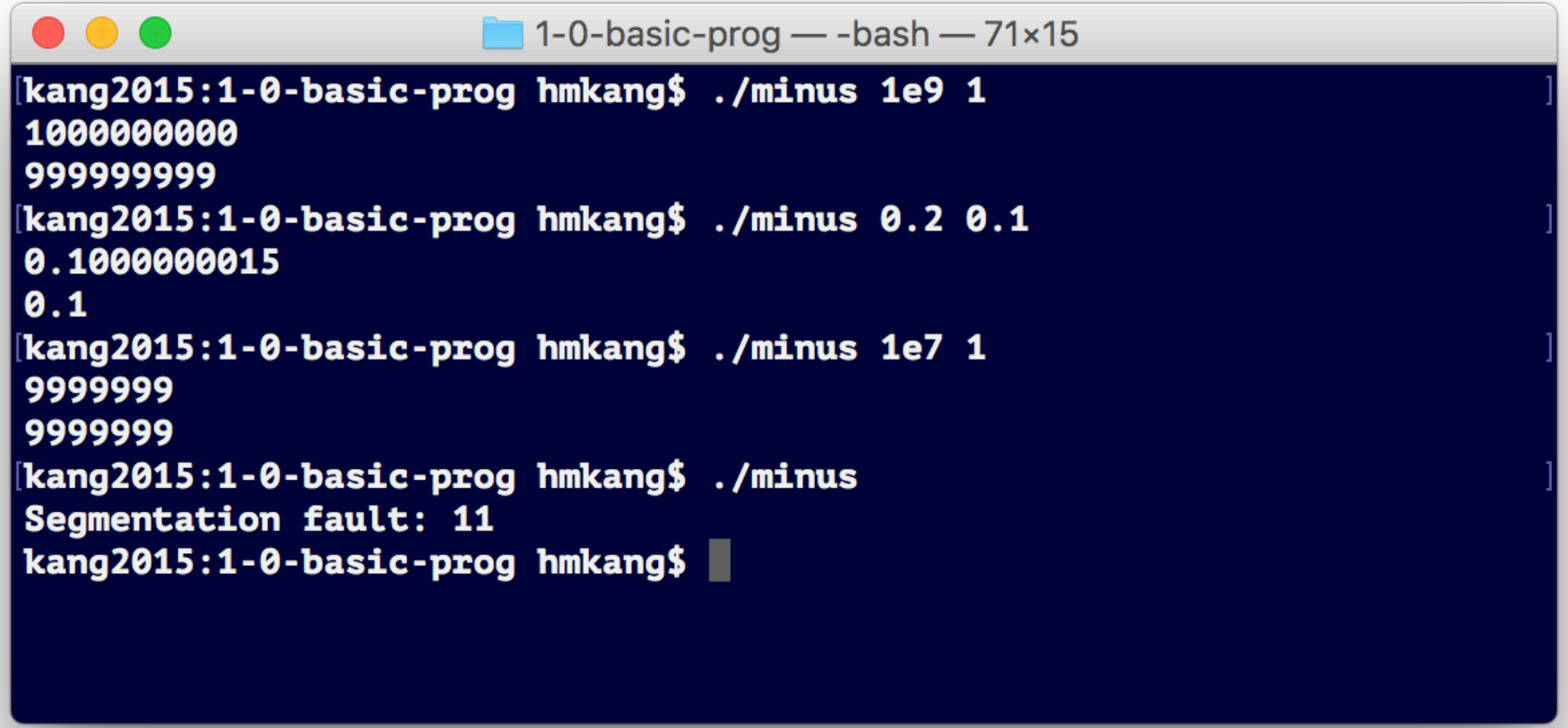
  First, include "`#include <cstdlib>`" before converting.

```
float f = strtof(s, NULL);
double d = strtod(s, NULL);
int i = strtol(s, NULL);
```

Implementing a flexible `minus` program

```
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4  int main(int argc, char** argv) {
5    float a = strtof(argv[1],NULL), b = strtof(argv[2],NULL);
6    float c = a - b;
7
8    double d = strtod(argv[1],NULL), e = strtod(argv[2],NULL);
9    double f = d - e;
10
11   cout.precision(10);
12   cout << c << endl;
13   cout << f << endl;
14
15   return 0;
16 }
17
```

Example output



Robust handling of arguments

- **If two arguments are given:**
  - Use the command-line arguments as input values

- **Otherwise**
  - Request to type two numbers via keyboard
  - Use those two numbers as input values

Robustly handling input data

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <string>
4  using namespace std;
5  int main(int argc, char** argv) {
6    string arg1, arg2;
7    if ( argc != 3 ) {
8      cout << "Type two numbers: ";
9      cin >> arg1 >> arg2;
10   }
11   else {
12     arg1 = argv[1]; arg2 = argv[2];
13   }
```

Using `std::string` instead of `char*`

- **C++ way to represent a string:**
  First, include "`#include <string>`" before using it.
  - `std::string s = "Hello";`

- **Converting C++-style string to (read-only) C-style string**
  - `const char* s2 = s.c_str();`

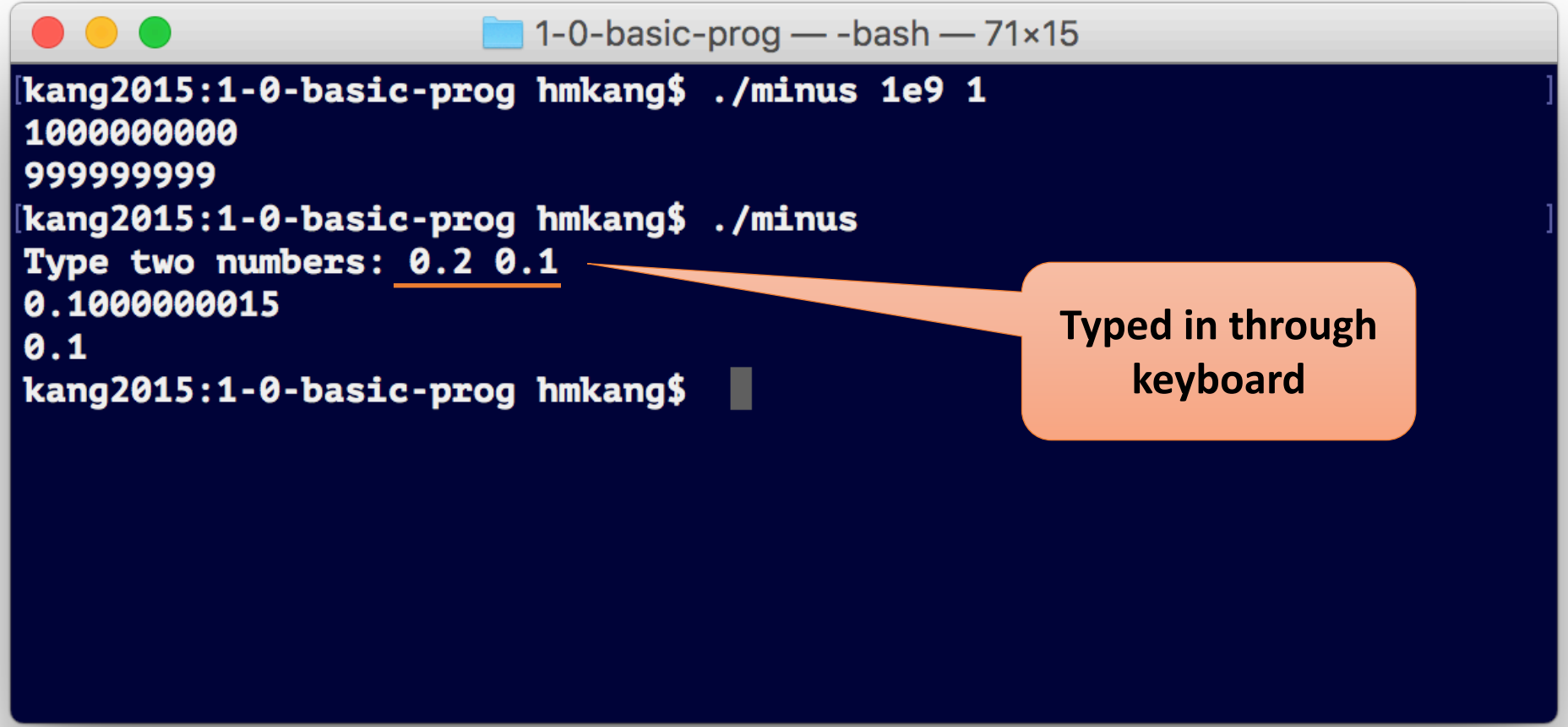
- **Converting C++ string to numeric numbers**
  First, include "`#include <cstdlib>`" before converting.

```
float f = strtof(s.c_str(), NULL);
double d = strtod(s.c_str(), NULL);
int i = strtol(s.c_str(), NULL);
```

Modified code for conversion & printing

```
15    float a = strtof(arg1.c_str(),NULL), b = strtof(arg2.c_str(),NULL);
16    float c = a - b;
17
18    double d = strtod(arg1.c_str(),NULL), e = strtod(arg2.c_str(),NULL);
19    double f = d - e;
20
21    cout.precision(10);
22    cout << c << endl;
23    cout << f << endl;
24
25    return 0;
26  }
```

Running example



Challenge:

Can you implement a similar program in python and R?

A python implementation

```
1  import sys
2  if ( len(sys.argv) != 3 ):
3      print("Please enter two numeric arguments")
4  else:
5      a = float(sys.argv[1])
6      b = float(sys.argv[2])
7      c = a - b
8      print(c)
9
```

An R implementation

```
1  args = commandArgs(trailingOnly = TRUE)
2  if ( length(args) != 2 ) {
3    stop("Please enter two numeric arguments")
4  } else {
5    options(digits=10)
6    cat(as.numeric(args[1])-as.numeric(args[2]))
7    cat("\n")
8  }
9
```

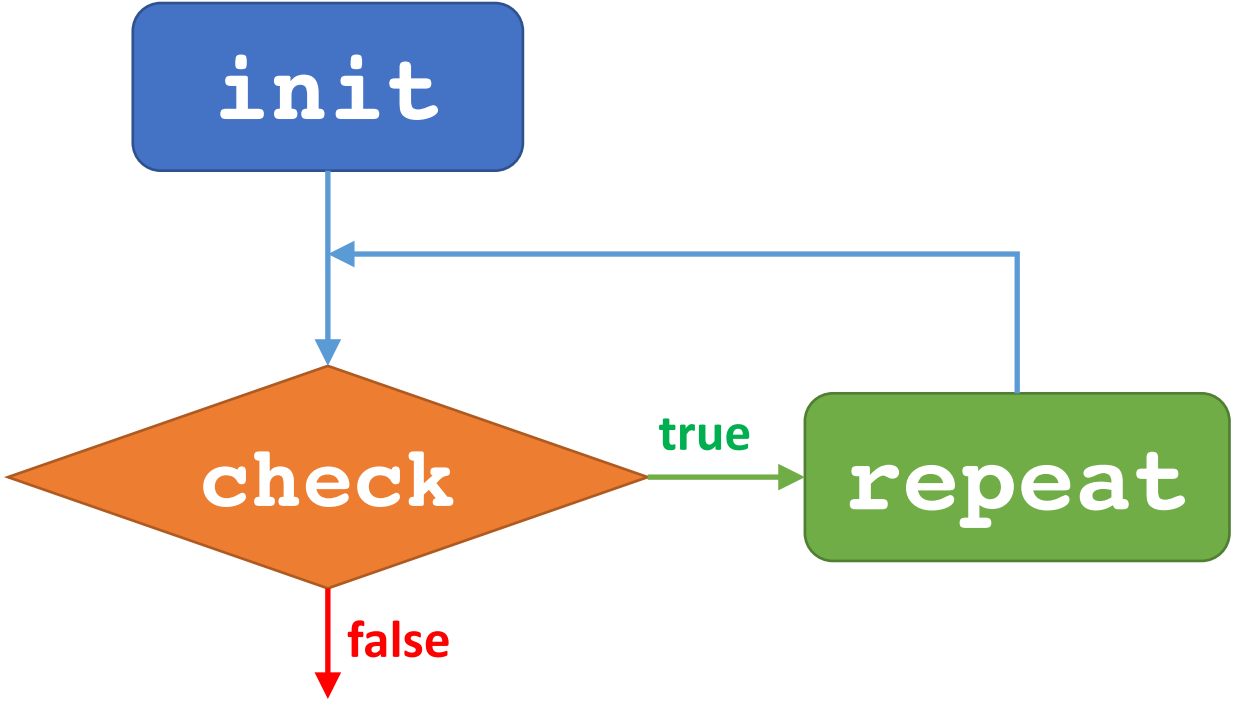
An application: Computing π numerically

- **Fact:** $$\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$$

$$\pi = \sqrt{6 \sum_{i=1}^{\infty} \frac{1}{i^2}}$$

- **Does this work?**
- **How many digits can we calculate accurately?**

```
for(init; check; repeat) {..}
```



An implementation with single precision

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  int main(int argc, char** argv) {
5    int n = strtof(argv[1],NULL);
6    float sum = 0;
7    for(int i=1; i <= n; ++i) {
8      sum += (1.0/i/i);
9    }
10   cout.precision(10);
11   cout << sqrtf(sum * 6) << endl;
12   return 0;
13 }
```

need for **`sqrt(double)`, `sqrt(float)`**

- Will this work?
- Alternative way to do better than this?

A running example

```
kang2015:1-0-basic-prog hmkang$ ./pi 100
3.13207674
kang2015:1-0-basic-prog hmkang$ ./pi 1000
3.140638351
kang2015:1-0-basic-prog hmkang$ ./pi 3000
3.141268492
kang2015:1-0-basic-prog hmkang$ ./pi 5000
3.141393423
kang2015:1-0-basic-prog hmkang$ ./pi 10000
3.141393423
kang2015:1-0-basic-prog hmkang$
```

- How many digits are correct?
- Why do 5,000 and 10,000 give the same answer?

```cpp
#include <iostream>
#include <cmath>
using namespace std;
int main(int argc, char** argv) {
  int n = strtof(argv[1],NULL);
  float ffsum = 0, frsum = 0;
  double dfsum = 0, drsum = 0;
  for(int i=1; i <= n; ++i) {
    ffsum += (1.0/i/i);
    frsum += (1.0/(n-i+1)/(n-i+1));
    dfsum += (1.0/i/i);
    drsum += (1.0/(n-i+1)/(n-i+1));
  }
  cout.precision(10);
  cout << sqrtf(ffsum * 6) << endl;
  cout << sqrtf(frsum * 6) << endl;
  cout << sqrt(dfsum * 6) << endl;
  cout << sqrt(drsum * 6) << endl;
  return 0;
}
```

Changing orders and precisions

Forward vs. Reverse sum

Single vs. Double precision.

- Which ones are better?
- By how much?

A running example

```
kang2015:1-0-basic-prog hmkang$ ./pi2 10000
3.141393423
3.141497135
3.141497164
3.141497164
kang2015:1-0-basic-prog hmkang$ ./pi2 2000000000
3.141393423
3.141592503
3.141592645
3.141592653
kang2015:1-0-basic-prog hmkang$
```

- What are your answers to the previous questions?

Summary

- Compiler & Interpreter languages are different
- Bits, bytes, and fundamental data types in C++
- Precision of floating-point values
- Using `<iostream>` for standard I/Os
- Command line arguments
- String conversions to numeric values
- Python & R implementations
- Using for loops