MODULE 1 / UNIT 4

FILE I/O, ARRAY, STRING, AND VECTOR IN C++

Today

- Time complexity of mergesort
- Introduction to STL : Standard Template Library
- File I/Os using STL : `#include <fstream>`
- Arrays and pointers
- Strings in C and C++
- Abstract data types
- Container types in C++ template libraries
  - `std::vector<T>`

Time complexity of mergesort

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + c2^k \\
&= 4T(2^{k-2}) + 2c2^k \\
&= 8T(2^{k-3}) + 3c2^k \\
&= \dots \\
&= 2^k T(1) + kc2^k
\end{aligned}
$$

Time complexity of mergesort

$$2^k \leq n < 2^{k+1}$$

$$2^k T(1) + kc2^k \leq T(n) \leq 2^{k+1}T(1) + (k+1)c2^{k+1}$$

$$\frac{n}{2}T(1) + c\frac{n}{2}\log\left(\frac{n}{2}\right) \leq T(n) \leq 2nT(1) + 2cn\log(2n)$$

$$\Theta(n\log n) \leq T(n) \leq \Theta(n\log n)$$

Using `NumericVector` in `Rcpp`

- **`NumericVector`** is a vector type passed to C++ from R

```
NumericVector x(10); // create a size 10 vector
unsigned int n = x.size(); // get the size of vector
x[0] += 2.0; // add 2.0 to the first element
```

- **`Cloning a NumericVector`** fully or partially

```
NumericVector y(x.begin(), x.end());
NumericVector z(x.begin()+10, x.begin()+20);
```

Standard Template Library (STL)

- **A software library for C++ programming language.**
- **A part of C++ standard – included in all C++ compilers**
- **Provide a commonly used data structure that could be complicated to implement otherwise.**
  - Input/output streams
  - Strings
  - Containers
  - Iterators
  - ....

Using <fstream> for file read/write

```
#include <Rcpp.h>
#include <fstream>
using namespace Rcpp;
using namespace std;

// [[Rcpp::export]]
int countWords(string filename) {
  ifstream ifs(filename);
  string s;
  int count = 0;
  while ( ifs >> s )
    ++count;
  ifs.close();
  return count;
}
```

Running the C++ code inside R

```
countWords('dolch.314.txt')
```

```
[1] 314
```

DIY R Notebook

`bios615_1_4.Rmd`

section #1

Arrays in C/C++

- Conceptually, an array should be a collection of elements with a certain data type.
- In C++, the syntax appears to so, for example:

```
int primes[10] = {2,3,5,7,11,13,17,19,23,29};
for(int i=0; i < 10; ++i)
  cout << primes[i] << endl;
```

- But in fact, the array in C/C++ represent an "address"

Address	d190
Value	46f0

primes

Address	46f0	46f4	46f8	46fc	4700	4704	4708	470c	4710	4714
Value	2	3	5	7	11	13	17	19	23	29

Printing addresses and values of array

```
#include <Rcpp.h>
#include <iostream> // needed for std::cout
#include <iomanip>  // needed for std::hex, std::dec
using namespace Rcpp;
using namespace std;
// [[Rcpp::export]]
void arrayTest() {
  int primes[10] = {2,3,5,7,11,13,17,19,23,29};
  for(int i=0; i < 10; ++i) {
    cout << "&primes[" << i << "] = " << hex << &primes[i] << "\t";
    cout << "primes[" << i << "] = " << dec << primes[i] << endl;
  }
  cout << "primes = " << hex << primes << endl;
}
```

primes == &primes[0]

```
arrayTest()
```

```
&primes[0] = 0x7fff5fbfbf80 primes[0] = 2
&primes[1] = 0x7fff5fbfbf84 primes[1] = 3
&primes[2] = 0x7fff5fbfbf88 primes[2] = 5
&primes[3] = 0x7fff5fbfbf8c primes[3] = 7
&primes[4] = 0x7fff5fbfbf90 primes[4] = 11
&primes[5] = 0x7fff5fbfbf94 primes[5] = 13
&primes[6] = 0x7fff5fbfbf98 primes[6] = 17
&primes[7] = 0x7fff5fbfbf9c primes[7] = 19
&primes[8] = 0x7fff5fbfbfa0 primes[8] = 23
&primes[9] = 0x7fff5fbfbfa4 primes[9] = 29
primes = 0x7fff5fbfbf80
```

DIY R Notebook

`bios615_1_4.Rmd`

section #2

Using `printf()` function

- **A C-style alternative to `std::cout` is `printf()`**
- **Need to include `<cstdio>` to use `printf()` function**
- **Advantages of `std::cout` and `operator<<`**
  - Output can be customized by each data types
  - Does not need to specify the data type
  - Works beyond built-in type (if `<<` operator is defined)
- **Advantages of `printf()`**
  - Easier to understand what the will be printed exactly
  - Multiple ways to print the same variable.
  - Easier to provide tailored-format output.

Specification of `printf()` function

- **Function can take variable-size arguments.**

```
printf( "string with %x %d ...", arg1,..., argn)
```

- **First argument:**
  - A single string, with the some special tokens

`%c`	(unsigned) char	`%o`	unsigned octal int	`%s`	(const) char* : string	`%[l]e`	scientific
`%d`	(signed) int	`%x`	unsigned hex int	`%f`	fixed-point float	`%[l]g`	fixed-point+ scientific
`%u`	unsigned int	`%p`	pointer to void*	`%lf`	fixed-point double		

- **Second to the last arguments**
  - Values for each special tokens in order.

Examples using `printf()`

```
#include <Rcpp.h>
#include <cstdio>
// [[Rcpp::export]]
void printfTest() {
  float f = 0.1 + 0.2;
  double d = 0.1 + 0.2;
  printf("Printing integer %d, and unsigned integer %u\n", -1, -1);
  printf("Printing decimal %d, octal %o, hex %x integers\n", 33, 33, 33);
  printf("Printing the address of f %p and d %p variables\n", &f, &d);
  printf("Printing a character as is %c or as an ASCII code %u\n", 'H', 'H');
  printf("Printing values of f %f and d %lf in a fixed-point format\n", f, d);
  printf("Printing values of f %e and d %le in a scientific format\n", f, d);
  printf("Printing values of f %g and d %lg in a flexible format\n", f, d);
  printf("Printing values of f %08.20f and\n d %08.20lf in an "
         "advanced fixed-point format\n", f, d);
}
```

Expected outcomes

```
printfTest()
```

```
Printing integer -1, and unsigned integer 4294967295
Printing decimal 33, octal 41, hex 21 integers
Printing the address of f 0x7fff5fbfbfdc and d 0x7fff5fbfbfd0 variables
Printing a character as is H or as an ASCII code 72
Printing values of f 0.300000 and d 0.300000 in a fixed-point format
Printing values of f 3.000000e-01 and d 3.000000e-01 in a scientific format
Printing values of f 0.3 and d 0.3 in a flexible format
Printing values of f 0.30000001192092895508 and
 d 0.30000000000000004441 in an advanced fixed-point format
```

DIY R Notebook

`bios615_1_4.Rmd`

section #3

std::string vs. (const) char*

- **C style string (`char*`)**
  - It represents an array of characters (i.e. a string)
  - But in fact it is just the address where the first character is stored.

```
const char* s1 = "Hello";
char s2[6] = {'H','e','l','l','o','\0'};
```

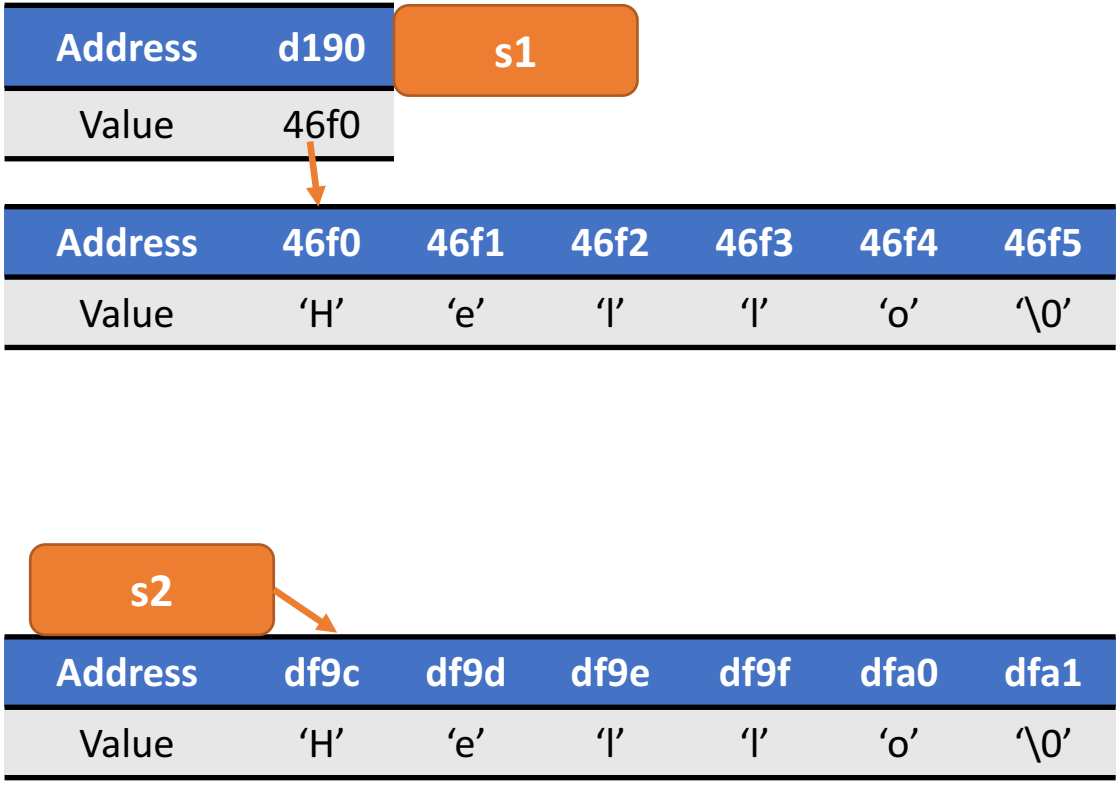
- **C++ style string (`std::string`)**
  - It behaves like an "object" rather than an "address"
  - Automatically converted from C style string (i.e. address)
  - Must be manually converted to C style string, if needed.

ASCII Code Chart

ASCII Code Chart

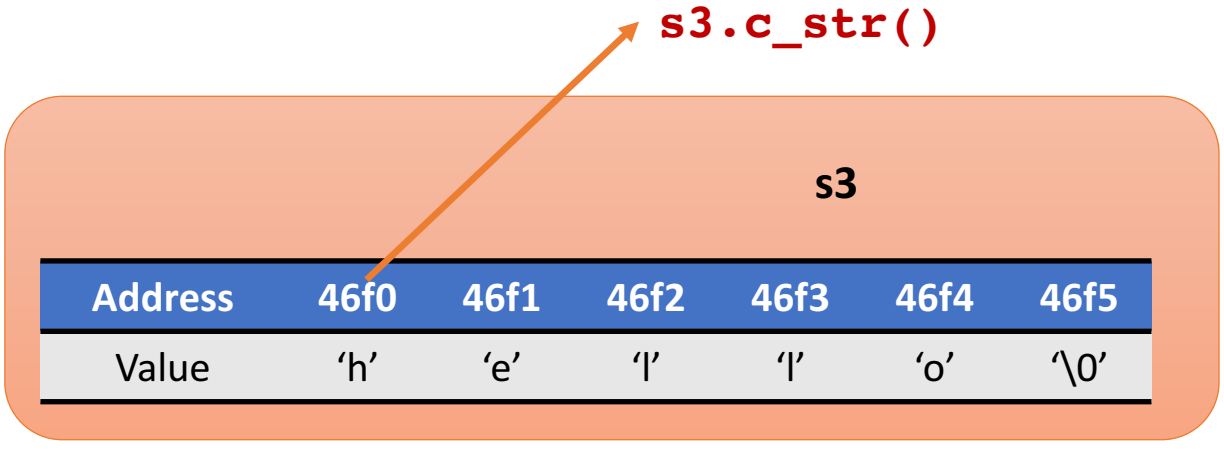
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Example memory structure with `char*`



Example memory structure with `std::string`

- `std::string s3("Hello")`
  - Equivalent to `std::string s3 = "Hello"`



```cpp
#include <Rcpp.h>
#include <cstdio> // needed for std::cout
#include <string> // needed for std::string
using namespace std;
// [[Rcpp::export]]
void stringTest() {
  const char* s1 = "Hello";
  char s2[6] = {'H','e','l','l','o','\0'};
  string s3("Hello");
  const char* s4 = s2;
  const char* s5 = s3.c_str();

  printf("%p\t%p\t%c\n", &s1, s1, s1[0]);
  printf("%p\t%p\t%c\n", &s2, s2, s2[0]);
  printf("%p\t%p\t%c\n", &s3, s3.c_str(), s3[0]);
  printf("%p\t%p\t%c\n", &s4, s4, s4[0]);
  printf("%p\t%p\t%c\n", &s5, s5, s5[0]);
}
```

Examples using `char*` and `std::string`

An example output

```
stringTest()
```

```
0x7fff5fbfbf78  0x10c734b80 H
0x7fff5fbfbfb0  0x7fff5fbfbfb0  H
0x7fff5fbfbf90  0x7fff5fbfbf91  H
0x7fff5fbfbf80  0x7fff5fbfbfb0  H
0x7fff5fbfbf88  0x7fff5fbfbf91  H
```

DIY R Notebook

`bios615_1_4.Rmd`

section #4

Abstract Data Types (ADT)

- Mathematical model for data types
- Its behavior (semantics) is defined by a set of values and operations.
- Theoretical concept that allows algorithms to separate from a particular method of implementation.

Example Container ADTs

Sensitive to input orders

- Stack
- Queue
- List
- Priority queue

Insensitive to input orders

- Set
- Map

Example Container ADTs

Sensitive to input orders

- Stack : LIFO

  `push, pop`

- Queue : FIFO

  `enqueue, dequeue`

- List : random access

  `front, back, next,`
  `insert, remove`

- Priority queue

  `insert_with_priority,`
  `pop_highest_priority`

Insensitive to input orders

- Set : key only

  `insert, remove, has_key`

  `front, back, next`

- Map : (key, value) pair

  `insert, remove`

  `has_key, get_value`

  `front, back, next`

Data structure

- **A specific way to organize the data in a computer.**

- **Key factors : correct and efficient algorithms**
  - .. to store values and perform operations

- **STL containers include several data structures implementing ADTs**

Container data structure in C++ STL

Sensitive to input orders

- Stack : `std::stack`
- Queue : `std::queue`
  `std::deque`
- List : `std::list`
  `std::vector`
- Priority queue
  `std::priority_queue`

Insensitive to input orders

- Set : `std::set`
  `std::unordered_set`
- Map : `std::map`
  `std::unordered_map`

Using `std::vector<T>`

- STL vector is a flexible-sized array that can contain an arbitrary type.
- Because it is using "template", the data type of the elements must be specified in definition

```
std::vector<std::string> example_str_array;
std::vector<double> example_dbl_array(10, 0);
```

- Similar to C-style array, elements can be access using `operator[]`
- A new element can be appended using `push_back()` function
- The size of array can be changed using `resize()` function
- See more at http://www.cplusplus.com/reference/vector/vector/

An example code

```
#include <Rcpp.h>
#include <fstream>  // need to use std::ifstream
#include <iostream> // need to use std::cout
#include <vector>   // need to use std::vector
using namespace Rcpp;
using namespace std;

// [[Rcpp::export]]
void loadWords(string filename) {
  ifstream ifs(filename);
  string s;
  vector<string> vecstr; // a vector of string
  while ( ifs >> s )
    vecstr.push_back(s);
  ifs.close();
  cout << "Finished loading " << vecstr.size() << " words" << endl;
  cout << "The first word is " << vecstr[0] << endl;
  cout << "The last word is " << vecstr[vecstr.size()-1] << endl;
  cout << "The word in the middle is " << vecstr[vecstr.size()/2] << endl;
}
```

An example output

```
loadWords('dolch.314.txt')
```

```
Finished loading 314 words
The first word is a
The last word is your
The word in the middle is like
```

```
loadWords('common.2198.words.txt')
```

```
Finished loading 2198 words
The first word is a
The last word is zone
The word in the middle is likely
```

```
loadWords('mit.10000.words.txt')
```

```
Finished loading 10000 words
The first word is a
The last word is zus
The word in the middle is lanka
```

DIY R Notebook

`bios615_1_4.Rmd`

section #5

Summary

- `std::ifstream` in `<fstream>`
- C-style arrays
- C-style `printf()` in `<cstdio>`
- `std::string` in `<string>` and C-style `char*` and arrays
- Abstract Data Types
- `std::vector<T>` in `<vector>`