Murdoch
UNIVERSITY

Data Structures and Abstractions

Arrays & Records & Strings & STL

Lecture 3

- The revision exercise must be completed before starting on Topic 3.

Arrays in C++

- Arrays in C++ are not a predefined type, they have to be defined by the programmer: [1]

```
const int SIZE = 10; [2]
...
typedef int ArrayType[SIZE];
...
ArrayType numbers;
```

- This declares an array of 10 integer point numbers with indexes from 0 to 9.
- Processing of arrays is the same in C++ as in C or Java, with [ ] giving access to elements.

Within Memory

- Within memory, the compiler has generated code to allocate 10 contiguous slots on the stack, each large enough to store one floating point number.

- The variable '**`numbers`**', is in fact a "*pointer"* to the first of the 10 memory locations. But you cannot put a new address into it. What are the consequences if you were able to do this?

- This can be seen if you output the array variable without an index:

  ```
  cout << hex << numbers << endl
  ```

  would give output something like: **`0x23ab34`** `[1]`

  which is the address in RAM of the 0th location out of the 10 integers.

- However,

  ```
  cout << numbers[0] << endl;
  ```

  will output the contents of the 0th memory location.

```
// Arrays1.cpp
// Demonstrating array use
// Version: 01, Nicola Ritter
// Version: 02, SMR
//-------------------------------------------------------------

#include <iostream>

using namespace std;

//-------------------------------------------------------------

const int SIZE = 10;
const int END = -1;

//-------------------------------------------------------------

// Define a new type that is an array of SIZE integers
typedef int IntArray[SIZE];
```

```
//----------------------------------------------------

// Returns the number of input values
//This is called a forward declaration
// Both the lines shown next are examples of forward declaration
int Input (IntArray &array);
void Output (int size, const IntArray &array);
// Why const?

//----------------------------------------------------

int main()
{
        IntArray array;

        int size = Input (array);  // size returns how many
        Output (size, array);  // this is a procedure

        return 0;
}

//----------------------------------------------------
```

```
int Input (IntArray &array)
{
    cout << "Enter up to " << SIZE
        << " positive integers, pressing <Enter> after each integer"
        << endl << "Use " << END << " to end input."
        << endl;

    int num;
    int index;
    cout << "Enter integer: ";
    cin >> num;
    for (index = 0; index < SIZE && num != END; index++)
    { // is the for loop the correct loop to use for this situation?
        array[index] = num;
        cout << "Enter integer: ";
        cin >> num;
    }

    return index;
}
```

```
//----------------------------------------------------------

void Output (int size, const IntArray &array)
{
    cout << endl << "Array at address " << array << " contains:"
     << endl;
    for (int index = 0; index < size; index++)
    {
      cout << index << ") " << array[index]
        << " at memory location " << &(array[index])
        << endl;
    }
    cout << endl;
}

//----------------------------------------------------------
```

Two Dimensional Arrays

- Arrays of more than one dimension are defined in a similar way:

```
float
twoDimArray[ROWS][COLS];
```

- Access is as per normal:

```
twoDimArray[row][col];
```

Dynamic Arrays

- Arrays can be declared dynamically as well:

  ```
  int *array = new int[size];
  ```

  declares an array of ‘size’ integers.

- As it is a dynamic declaration, the integers are on the *heap* rather than the stack. Don’t forget to delete them afterwards when you no longer need them.

- If there is not enough memory (new fails), array is set to NULL (or the preferred **nullptr**).

- However use of the array does not change unless it was set to NULL. What would happen if it was set to NULL?

```
// TwoDimArray.cpp
// Program designed to show the declaration and use of a dynamically
//   created two dimensional array
//
// Version
// 001 First Attempt, Nicola Ritter
// modified smr
//---------------------------------------------------------------

#include <iostream>
#include <iomanip>
using namespace std;

//---------------------------------------------------------------

const int COL_WIDTH = 14;

//------------------Forward declarations---------------------------

void GetSizes (int &rows, int &cols);
float **CreateArray (int rows, int cols);
void Output (int rows, int cols, float **array);
void Initialise (int rows, int cols, float **array);
void Delete (int rows, float **&array);

//---------------------------------------------------------------
```

```
int main ()
{
    // Declare a pointer to an array of pointers
    float **array = NULL;
    int rows, cols;

    GetSizes (rows, cols);

    // Get memory for the 2 dim array
    array = CreateArray (rows, cols);

    if (array != NULL)
    {
        // Set the table to contain 0s
        Initialise (rows, cols, array);

        // Output to check
        Output (rows, cols, array);

        Delete (rows, array); // this is a user defined delete, not C++ delete [1]
    }

    // put in an else part which says that memory for array could not be created.

    return 0;
}
```

```
//--------------------------------------------------------------------

void GetSizes (int &rows, int &cols)
{
    cout << "How many rows do you want? ";
    cin >> rows;
    cout << "How many columns do you want? ";
    cin >> cols;
    cout << endl;
}

//--------------------------------------------------------------------
```

```
float **CreateArray (int rows, int cols)
{
    // Get an array of pointers for the rows
    float **array = new float* [rows]; //draw this structure

    if (array != NULL) // how about an else
    {
        // Now get a row for each of these pointers
        for (int index = 0; index < rows; index++)
        {
            array[index] = new float[cols];
        }

        // Check that allocation occurred
        if (array[rows-1] == NULL)
        {
            Delete (rows, array);
        }
    }

    return array;
}
```

```cpp
//-----------------------------------------------------------------

void Output (int rows, int cols, float **array)
{
    cout << "The array values are:" << endl;
    for (int row = 0; row < rows; row++)
    {
        for (int col = 0; col < cols; col++)
        {
            cout << setw(COL_WIDTH) << array[row][col];
        }
        cout << endl;
    }
    cout << endl;
}

//-----------------------------------------------------------------
```

```
•   void Initialise (int rows, int cols, float **array)
•   {
•       for (int row = 0; row < rows; row++)
•       {
•           for (int col = 0; col < cols; col++)
•           {
•             array[row][col] = 0;
•           }
•       }
•   }

•   //-------------------------------------------------------------------
```

```
void Delete (int rows, float**  &array)
{
        for (int row = 0; row < rows; row++)
        {
                if (array[row] != NULL)
                {
                   delete [] array[row]; // draw a diagram
                }
        }

        delete [] array;
        array = NULL;
}

//-----------------------------------------------------------
```

Input Testing

- When testing a program it can become onerous to keep having to type in data.

- For this reason it is common to store test data in a file and run the program from the command line, redirecting data into the program:

```
aprogram.exe < data.txt
```

- The data stored in **`data.txt`** will be read into the program *as if it had been entered from the keyboard.* [1]

- Note that this is *not* the same as file input, it is a way to replicate keyboard typing, without having to do it again and again.

- Note that the output data can also be redirected to file:

```
aprogram.exe < data.txt > data.out
```

Murdoch
UNIVERSITY

```
// redirect.cpp
// Testing redirected input from a file

#include <iostream>

using namespace std;

int main ()
{
        int number;

        do
        {
                cin >> number;
                cout << number << endl;

[1]     } while (cin.good());

        return 0;
}
```

Stops the loop when the redirected file reaches eof

Input File (data.txt)

```
2
45
67
8
912
```

User types in [2]

```
redirect < data.txt
```

Screen Output

```
2
45
67
8
912
```

Records

- Records (grouped data) are produced in C and hence C++ using the **`struct`** type:

```
typedef struct Person
{
 char firstName[SIZE];
 char surname [SIZE];
 int  age;
};
```

- A variable of this type is then declared in the same way as any other type:

```
Person person; [1]
```

- Access to parts of the person is done using the dot operator:

```
cout << person.firstName << " "
     << person.surname << " "
     << person.age << endl;
```

An Array of Records

- Similarly, we could declare an array of records as:

```
Person people[ARRAY_SIZE];
```

- And access an individual part of a single record:

```
people[index].firstname
```

Readings

- Textbook by Malik
  - Chapter on Arrays and Strings
  - Chapter on Records

What is the STL? [1]

- When C++ was written it was essentially a superset of C that allowed object oriented programming.
- At the start people either used the C-style arrays and strings (that we have already looked at and revised), or created their own classes.
- This was patently silly, so in the mid 1990s a standard set of templates was created for commonly used structures that are containers of data. [2]
- A container stores multiple objects of the same type. What changes between different types of container is the type of access, ordering and methods available.
- In this unit we will examine some of the most commonly used STL containers.

Problems with the STL

- The main problem with the STL is that it was clearly written by a committee.
- This means that:
  - It is ‘clunky’ – tries to please everyone.
  - Method names are often verbose, so everyone knows.
  - Use of methods may not intuitive all the time.
  - No bounds checking is done on arrays (vectors)!
  - The names of methods and classes is often non-mainstream. [1]
  - *Iterators* — which are generalisation of pointers — are used for a lot of the time, rather than indexes.
  - As it was written after C++ it might not always interface well with C++ and not all versions of C++ support it. [2]
  - But it is stable and usually well debugged – so can be used and usually very trusted.

The standard string class

- ANSI/ISO standard for C++ requires a string class to be a basic data type.
- Programmer should not have to worry about implementation details of the string class.
- It is part of the std namespace. [1]
- Is not part of C++ (as in reserved word); it is a programmer defined type available in the standard library.
- It has a wealth of good functions that make string handling much simpler.
- It has overloaded operators so that you can do such things as add to a string using the + operator.
- It makes reading in a whole line of text (including spaces) trivial.
- To use the standard string you must include the header file **`<string>`**.

The std string

- Unlike a C-style string, an std string cannot overflow.
- However no bounds checking is done on access.
- Do not assume that std string is NULL terminated. [1]
- Access to the string can be done as per any other string, using the [] operator.
- There are many good websites giving information on the various aspects of C++ including the std string.
- The following site is recommended. http://www.cppreference.com/cppstring/index.html

String Operator List

In each of the examples below, str1, str3 and str4 must be std strings, however str2 can be a c-style string. // std:string str1, str3, str4 [1]

`str1 = str2`	Copies str2 to str1

`str1 == str2`
`str1 < str2`
`str1 > str2`
	These give true or false depending on whether str1 is lexicographically equivalent to, less than or greater than str2.

`str1 = str3 + str4`	str1 becomes the concatenation of str3 and str4.
`str1 += str2`	str2 is appended to str1
`str1 += ch`	The character ch is appended to str1
`cout << str1`	str1 is output to screen
`cin >> str1`	The first word typed at the keyboard is stored in str1

Std::string methods

- Each method is overloaded multiple times.
- For example, to append to a string you could use:

`str1.append (str2)`	Appends str2 to the end of str1, where str2 can be a c-style OR std string.
`str1.append(str2, index, num)`	Append num characters from the part of str2 that starts at index.
`str1.append (str2, num)`	Append num characters from the start of str2, where str2 can be a c-style or std string.
`str1.append (num, ch)`	Add num repetitions of ch to the end of str1.
`str1.append (itr1, itr2)`	Append the string that starts at iterator itr1 and ends at iterator itr2.

List of Commonly Used Methods

- <various> below indicates that the method is overloaded and there are various possible parameters.
- For more information, go to http://www.cppreference.com/cppstring/index.html or your text book.

`str1.append (<various>)`	Append values str1.
`str1.c_str ()`	Returns a standard c-style string (char *). Required for file opening etc.
`str1.clear ()`	Empties the string.
`str1.compare (<various>)`	Compares two strings.
`str1.copy (<various>)`	Copies data into str1.
`str1.empty ()`	Returns true if the string is empty.
`str1.erase (<various>)`	Erases a part of the string.

Murdoch UNIVERSITY

Methods (continued)

`str1.find (<various>)`	Finds strings or characters within str1.
`str1.rfind (<various>)`	Finds strings/characters within str1, searching from the end.
`str1.insert (<various>)`	Inserts strings or characters into str1.

`str1.length ()`
`str1.size ()`
	Return the length of the string (number of objects in it)

`str1.push_back (ch)`	Add the character to the end of the string.
`str1.replace (<various>)`	Replace strings/characters within str1.
`str1.substr (index, num)`	Returns num characters starting from index.
`str1.swap (str2)`	Swaps the two strings.

Murdoch UNIVERSITY

The getline Function

- A really useful non-member string function.
- It allows the input of sentences—strings with white spaces—into a string:

```
string str;

// Read in one word only
cin >> str;

// Read in characters until eoln
getline (cin, str); [1]
```

- It is also possible to read in only part of a line, by specifying a delimiter. For example:

```
// Read in characters until a comma
getline (cin, str, ‘,’); [2]
```

Using Iterators

- All of the STL uses iterators for access to data. The same is true for std string.
- For this reason, there are two functions that are standard for each container in the STL:

`.begin()`	Returns an iterator pointing at the first object in the container.
`.end()`	Returns an iterator pointing at the first memory location *past* the end of the container.

Simple string access program

Uses the normal index type access method

```
•       #include <string>
•       #include <iostream>
•       using namespace std;

•       int main()
•       {
•               string str;

•               cout << "Enter a string: ";
•               getline (cin, str);

•               cout << endl
•             << "The characters in your string were: "
•             << endl;
•
•
        for (int index = 0; index < str.length(); index++)
        {
                cout << str[index] << endl;
        }
•               cout << endl;
•               return 0;
•       }
```

Simple string access program

Uses the iterator type access method

```
•         #include <string>
•         #include <iostream>
•         using namespace std;

•         int main()
•         {
•                 string str;

•                 cout << "Enter a string: ";
•                 getline (cin, str);

•                 cout << endl
•               << "The characters in your string were: "
•               << endl;
•
•
          for (string::iterator itr = str.begin();
                itr != str.end(); itr++)
          {
                cout << *itr << endl;
•         }       cout << endl;
•                 return 0;
•         }
```

Testing

- Always test for:
  - empty strings;
  - empty files;
  - incorrect file names;
  - read-only files.

- Only assume data is correct if you have it stated in writing.

- Never assume the user won’t make a mistake.

- Always have a test plan and run through it ***every*** ***time*** you alter the program in any way.

Text File I/O

- A name must be entered by the user.
- The file must be opened.
- A test must be done to check it has opened.
- Data is then read until EOF.

- File handling in C++ requires the **`<fstream>`** header file. See the lab exercises starting in topic 2.

```
// text.cpp
// Text file I/O
// Reading a file of integers separated by spaces or new lines
//
// Version
// 01 - Nicola Ritter
// 02 – modified smr
//-----------------------------------------------------------------

#include <iostream>
#include <fstream>
using namespace std;

//-----------------------------------------------------------------

const int SIZE = 256;
typedef char strType[SIZE+1];         // name the type

//-----------------------------------------------------------------
```

```
•    int main ()
•    {
•         strType filename;  // C array of chars

•         // Get a file name from the user
•         cout << "Enter name of file to read (one word only): ";
•         cin >> filename;

•         // Declare the file variable and try to open it using that
•         //   file name
•         ifstream fstr (filename);
```

```
        // Is the ready state 0 (no errors)?
        if (fstr.rdstate() == 0) //[1]
        {
            int number;

            // prime the while loop
            fstr >> number;
            while (!fstr.eof())
            {
                cout << number << endl;
                fstr >> number;
            }

        }
        else
        {
            cerr << "Could not open file \"" << filename
        << "\" for reading." << endl;
        }

        return 0;

    }
```

Opening Files using std string

```
string filename;
cout << “Enter filename: ”;
getline (cin, filename);

ifstream fstr (filename.c_str());

// The rest of the code is the
// same as before
```

File names can now contain spaces

`c_str()`

gives the required access to the actual character string

`str.c_str()`

can be also be used with some of the standard c-style string functions

More About File I/O

- Opening a file for output is done in a similar way, except that it will be an output file stream:
  `ofstream fstr (filename);`

- To append to a file, you would open it with:
  `ofstream fstr (filename, ios::app);`

- When you are going to store more of one type of data in a file, it is necessary to store the number of each type. You don't have to but this can cause problems in designing the algorithm. Algorithm would require asking the user or the value is hard coded.

- For example, when storing an array of 100 integers, you would write the 100 to the file first.

- When reading the file, you would then read the number 100 and therefore know that you had 100 integers to read.

Other Useful I/O Functions

- There are several generically useful I/O functions (as well as the formatting ones done previously) [1]:

`cin.good()`
`fstr.good()`
	Returns true if the stream (standard input or a file stream) has no errors.

`.flush()`	Flushes (empties) the stream (but not the actual file!).
`.peek()`	Peek at the next character without actually reading it.
`.ignore ()`	Ignore characters

Murdoch UNIVERSITY

Readings

- Chapter on Arrays and String, Section on C-Strings.
- Chapter on User-defined Simple Data Types, Namespaces, and the string type, section on string type.
- Chapter on Standard Template Library (STL)
- Website: http://www.cplusplus.com/faq/sequences/strings/split/
- Website: “What is an iterator in C++”, http://www.oreillynet.com/pub/a/network/2005/10/18/what-is-iterator-in-c-plus-plus.html - optional
- Video by Standford uni provided by academicearth. Links:
  - C/C++ Libraries
  - C++ Console I/O

Character Functions [1]

- There are many useful character functions available in C++.
- They are actually standard C functions.
- To use them use must include the header file **`<cctype>`**.
- Note that the original C header file was **`<ctype.h>:`** within C++ the ".h": is removed and a "c" added in front of the file name.
- Most of these functions query the classification of the character.
- Because they are C functions, not C++ functions, they return 0 for false and 1 for true.

Useful Character Functions

`isalpha(ch)`	Is it an alphabetic character?
`isascii(ch)`	Is it an ASCII character?
`isblank(ch)`	Is it a blank character?
`isdigit(ch)`	Is it a digit (0..9)?
`islower(ch)`	Is it a lowercase alphabetic character?
`isupper(ch)`	Is it an uppercase alphabetic character?
`ispunct(ch)`	Is it punctuation? In this sense punctuation is defined as any printable character that is not a space or an alphanumeric character.
`isspace(ch)`	Is it a space?
`isxdigit(ch)`	Is it a hexadecimal digit (a..f, A..F,0..9)?

Useful Character Functions cont.

There are two other useful functions:

`toupper (ch)`	If ‘ch’ is a lowercase character, return the uppercase character, or else return it unchanged.	`char ch = ‘a’;`
		`ch = toupper (ch);`

`tolower (ch)`	If ‘ch’ is an uppercase character, return the lowercase character, or else return it unchanged.	`char ch = ‘?’;`
		`ch = tolower (ch);`

Strings

- Strings are simply character arrays, and therefore defined the same way as for other arrays.
- However, strings must have a NULL character at their end, therefore if you want 20 characters, you must allow space for 21:

```
const int SIZE = 20;
...
typedef char StringType[SIZE+1];
...
StringType str;
```

- If you overflow your allocated space, strange things can happen. [1]

NULL

- The NULL character is the character with ASCII value 0:

```
char ch = 0;
```

- Or it can be designated as a character using a backslash:

```
char ch = ‘\0’;
```

- In fact many control and formatting characters have backslash equivalents, the most useful being:

newline	‘\n’
tab	‘\t’
quotes	‘\”’
backslash	‘\\’

Useful String Functions

- C (and hence C++) has one of the most powerful sets of string manipulation functions available.

- C String functions require the **`<cstring>`** header file to be included.
- The string functions *do no overflow checking!*
- The require null terminated char arrays.

String Function List

`int strlen (str)`	Returns the number of characters from the start of the string to the NULL character.
`int strcmp(str1,str2)`	Returns 0 if they are identical, a negative number if str1 is *lexographically* less than str2, and a positive number if str2 is greater than str1.
`int strncmp(str1,str2,n)`	Compares the first 'n' characters of each string. Returns 0 if they are identical, a negative number if str1 is *lexographically* less than str2, and a positive number if str2 is greater than str1.
`int strcpy(dest,src)`	Copies src to dest, but does no bounds check.
`int strncpy(dest,src,n)`	Copies the first 'n' characters from src to dest, does no bounds check.

String Function List Continued

`int strcat (dest, src)`	Copies src to the end of dest, does no bounds check.
`int strncat (dest, src, n)`	Copies the first ‘n’ characters of src to the end of dest, does no bounds check.
`char* strchr (str, ch)`	Searches str for the first occurrence of ch. Returns a *pointer* to ch, or NULL if the character does not occur.
`char* strrchr (str, ch)`	Searches str in for the last occurrence of ch. Returns a *pointer* to ch , or NULL if the character does not occur.

String Function List Continued

`char* strstr (haystack, needle)`	Searches haystack for needle. Returns a pointer to the position of needle within haystack, or NULL if the needle character string does not occur.
`char *strrstr (haystack, needle)`	Reverse search.
`int atoi (str1)`	Converts str1 to an integer and returns either the integer or 0 if str1 could not be converted to an integer.
`float atof (str1)`	Converts str1 to a floating point number and returns either the floating point number or 0 if str1 could not be converted to a float.
`itoa (value, str1, 10)`	Converts value to a base 10 string and puts it in str1. [1]

```
// strings1.cpp
// Simple program demonstrating the problems with overflow
// Version
// original by – Nicola Ritter
// modified smr

#include <cctype>
#include <iostream>
using namespace std;

const int SIZE = 5;
typedef char StringType[SIZE+1]; // why + 1

//------------------------------------------------------------
```

Code

```
int main()
{
stringType str1 = “abcde”;
stringType str2;

cout << "str1: " << str1 << endl;

strcpy(str2, “0123456789”);

cout << "str1: " << str1 << endl;

strcpy(str1, “vwxyz");

cout << "str1: " << str1 << endl;
cout << "str2: " << str2 << endl;

return 0;
}
```

str1	a	b	c	d	e	\0	▨	▨

Output

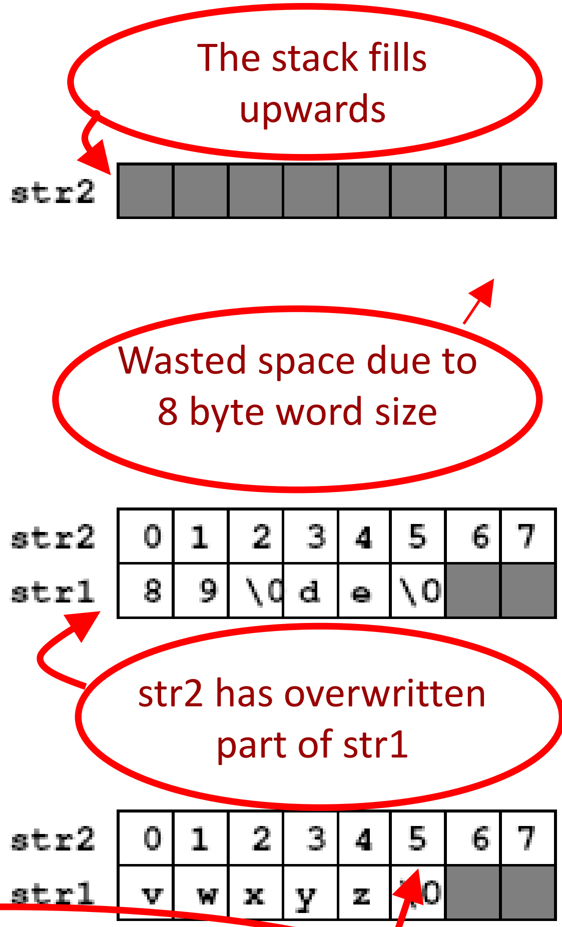
What output will we get?

```
str1: abcde

str1: 89

str1: vwxyz
str2: 01234567vwxyz
```

Memory



The stack fills upwards

str2	▨	▨	▨	▨	▨	▨	▨	▨

Wasted space due to 8 byte word size

str2	0	1	2	3	4	5	6	7
str1	8	9	\0	d	e	\0	▨	▨

str2 has overwritten part of str1

str2	0	1	2	3	4	5	6	7
str1	v	w	x	y	z	\0	▨	▨

We have fixed up str1, but str2 still has no ‘\0’ at the end of it.

Murdoch UNIVERSITY

Pointer Access to Strings

```
// Output one character per line

void CharOutput (const StringType str)
{
    // Set a pointer at the beginning of the string
    //   stop when it reaches the NULL character
    //   the increment points the pointer at the next character
    for (char *ptr = str; *ptr != ‘\0’; ptr++) [1] //use nullptr
    {
        // Output the character at ptr
        cout << *ptr << endl;
    }
}
```

Using Functions that Return Pointers

```
// Count the number of times str2 appears in str1

int Count (const StringType &str1, const StringType &str2)
{
        int counter = 0;

        // Prime the while loop by looking for a first occurrence
        char *ptr = strstr(str1, str2);

        // While we have found an occurrence of str2 in str1
        while (ptr != NULL)
        {
                counter++;
                ptr++; // go to the next char in str1. why? See next line
                ptr = strstr (ptr, str2);
        }

        return counter;
}
```

Readings

- Textbook Chapter on Arrays and Strings, section on C-Strings
- Online reference to C string with examples
  - http://en.cppreference.com/w/cpp/string/byte
  - http://www.cplusplus.com/reference/

- Reminder:
  - It is essential to read the specified readings listed above. The lecture notes are not a substitute for the readings as the lecture notes do not cover everything you must know. See advice in unit guide.