Introduction to C++

Pt. 3

Emanuele Giona Department of Computer Science, Sapienza University of Rome

Internet of Things A.Y. 2022/23

Prof. Chiara Petrioli Department of Computer, Control and Management Engineering, Sapienza University of Rome
Dott. Michele Nati

Exercise 1: solution

```cpp
#include <iostream>
using namespace std;

int main( ){
    int n = 0;
    cout << "Please insert an integer value:" << endl;
    cin >> n;

    int digitSum = 0;
    while(n > 0){
        digitSum += n % 10;
        n /= 10;
    }
    cout << "Sum of the number's digits is " << digitSum << endl;
    return 0;
}
```

Exercise 2: solution

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

void swapFloat(float &num1, float &num2);

int main(){
    float a, b;
    a = b = 0;

    cout << "Please insert two floating-point numbers:" << endl;
    cin >> a >> b;

    cout << "Value of 'a' is " << a << ", value of 'b' is " << b << endl;
    swapFloat(a, b);
    cout << "Value of 'a' is " << a << ", value of 'b' is " << b << endl;

    return 0;
}

void swapFloat(float &num1, float &num2){
    float tmp = num1;
    num1 = num2;
    num2 = tmp;
    return;
}
```

Exercise 3: solution

```cpp
#include <iostream>
using namespace std;

int main(){
    cout << "This program prints the first 10 natural numbers." << endl;
    int i = 0;

    statement: {
        cout << i << endl;
        i++;
    }

    checkCondition: {
        if(i < 10){
            goto statement;
        }
    }

    return 0;
}
```

1. Arrays

Arrays

An array is a data structure storing multiple variables of the same type. Typically, an index is used to access a specific variable within an array, representing its position.

There are many implementations of data structures with similar usage in C++:

- Static arrays
  Number of variables stored must be known at compile time
- Dynamically-allocated arrays
  Number of variables stored can be specified at runtime, maybe depending on user input
- Standard library (STL) containers
  Similar to dynamically-allocated arrays, but provide automatic memory management and typical data structure operations

Using a static array

Similarly to normal variables, arrays are declared with a type and an identifier:

```
int array[10];
char name[3] = {‘C’, ‘+’, ‘+’};
```

Square brackets [ ] are used to indicate how many variables can be stored in this array (10)

Array name can store at most 3 variables, initialized

Accessing arrays

Positions within arrays start from 0, and the access to a particular position relies on the array index operator [ ].

```
int array[10];
array[3] = 6;
array[9] = array[3] - 20;
```

Accessing an array past its maximum size will trigger a runtime error

0	1	2	3	4	5	6	7	8	9
			6						-14

All other indices are filled with default values, which depend on the type

2. Pointers

Low-level memory manipulation in C++

As previously mentioned, C++ allows for memory management both at a high level and a low level. Pointers are variables storing memory addresses of other variables, and can be used to manipulate memory at a low level.

Obtaining a variable's memory address

Assuming a variable `var` has already been declared, `&var` represents its memory address.

Accessing the referenced variable (dereferencing)

Assuming a pointer `pVar` to variable `var` has been declared, `*pVar` represents the referenced variable `var`.

Declaring a pointer

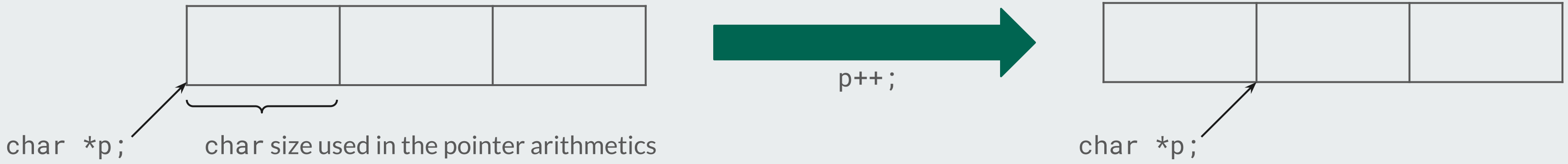
```
int var = 231;
int *pVar = &var;

if(*pVar == var)
  cout << "Pointing to var!" << endl;
```

Due to different metadata, pointers require specification regarding the type of the referenced variable
Do not confuse the asterisk * used in declaration from the one used during a dereferencing operation!

Pointer arithmetics

C++ only allows addition and subtraction operations on pointer variables, but with a special meaning: the stored memory address is modified based on the underlying type used by the referenced variable.



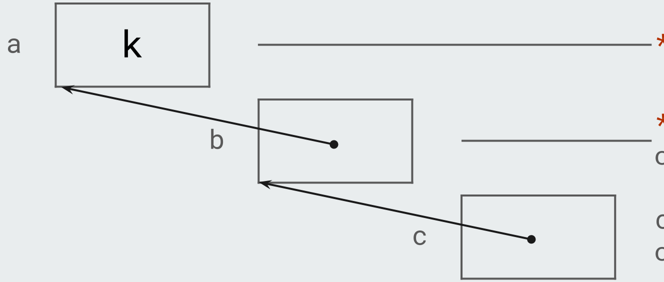
Increment (`++`) and decrement (`--`) operators have higher precedence than the dereferencing operator (`*`):

`*p++;`	→ Increment pointer, dereference previous address	*equivalent:* `*(p++);`
`*++p;`	→ Increment pointer, dereference new address	*equivalent:* `*(++p);`
`++*p;`	→ Dereference pointer, increment referenced variable's value	*equivalent:* `++(*p);`
`(*p)++;`	→ Dereference pointer, increment referenced variable's value	

Pointers to pointers

Multiple indirection levels are supported, and each of them requires an asterisk (*).

```
char a;
char *b;
char **c;
a = 'k';
b = &a;
c = &b;
```



**c (type char) contains value 'k'

c (type char) contains the memory address of a

c (type char**) contains the memory address of b

Pointers and arrays

Arrays can be implicitly converted to pointers, supporting the same set of operations on them. Indeed, the name of an array can be used to point to its first element.

```c
int array[5];
int *p;
p = array;
*p = 3;
*++p = 7;
p = &array[2];
*p = -5;
p = array + 3;
*p = 22;
array[4] = 1;
```

Resulting array

3	7	-5	22	1

Pointers and functions

Pointers can be passed as function arguments simply by declaring the required parameters as pointers.

Similarly to variables using pass-by-reference, changes on pointer variables inside the function will also affect the calling function.

```
#include <iostream>

void duplicate(int *array, int size);

int main(){
  int arr[3] = {1,3,5};
  duplicate(arr, 3);
  return 0;
}

void duplicate(int *array, int size){
  for(int i=0; i<size; i++, array++){
    *array *= 2;
  }
}
```

Beware assigning values to pointer variables within a function: in case of pointers to variables declared inside a function's body, such variable will be deleted upon function exit, leaving the pointer with an invalid memory address.
This is called a dangling pointer, causing severe issues if unchecked.

3. STL containers

Standard library containers

The C++ standard library (STL) provides many implementations of collections of elements called containers, as well as useful algorithms using them.

There are two main categories of containers:

- Sequential
- Associative

The container choice depends entirely on the functionality required, but computational complexity of the operations on a container may vastly differ from one another.

Sequential containers

In sequential containers, elements are stored in the same order as they are added to them and can be processed sequentially in the same order.

- STL containers
  `array`$^{C++11}$, `vector`, `list`, `forward_list`$^{C++11}$, `deque`
- STL adaptors
  `stack`, `queue`, `priority_queue`

Many containers share the same member functions and functionality, but computational complexity may differ. Adaptors are not entirely new containers, but rather build specific functionality on top of an existing container.

Sequential containers handbook

- `array`
  Pros: random access to elements, usage of STL containers' functions; Cons: fixed size
- `vector` or `deque`
  Pros: variable size, random access to elements, efficient for front or back operations; Cons: inefficient for operations at the middle
- `list` or `forward_list`
  Pros: variable size, efficient for operations at any point; Cons: memory overhead with small elements, inefficient for random access

Using STL `arrays` and `vectors`

```cpp
#include <iostream>
using namespace std;

int main(){
  // Automatic size based on
  // initialization value
  int arr[] = {4,52,23,6,9};
  for(int i=0; i<5; i++)
    cout << arr[i] << endl;
  return 0;
}
```

Built-in arrays require knowledge of size in order to iterate over them, whereas STL containers implement a `size()` function.
Additionally, many useful functions provided within the standard library only accept STL containers.

```cpp
#include <iostream>
#include <array>
using namespace std;

int main(){
  array<int,5> arr{4,52,23,6,9};
  for(int i=0; i<arr.size(); i++)
    cout << arr[i] << endl;
  return 0;
}
```

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main(){
  vector<int> arr{4,52,23,6,9};
  for(int i=0; i<arr.size(); i++)
    cout << arr[i] << endl;
  return 0;
}
```

Iterators

Iterators are objects that behave in a similar way to pointers, but specifically for STL containers.
They can be classified into the following categories:

- Input
- Output
- Forward
- Bidirectional
- Random-access

Not all containers support every type of iterator!

Utility of iterators

- Conveniency when working with containers
- Reusable interface for all containers
- Allow for dynamic insertion/deletion of elements

Iterators example

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
  vector<int> arr{4,52,23,6,9};
  for(int i=0; i<arr.size(); i++)
    cout << arr[i] << endl;
  return 0;
}
```

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
  vector<int> arr{4,52,23,6,9};
  vector<int>::iterator it;
  for(it=arr.begin(); it!=arr.end(); it++)
    cout << *it << endl;
  return 0;
}
```

Common iterator operations

- `begin(),end()`
  Return beginning / after-end positions of a container
- `advance()`
  Increments an iterator by a user-specified value
- `prev(),next()`
  Returns new iterators pointing at positions further back / forward by a user-specified value
- `inserter()`
  Inserts elements at any position of the container

Using `lists`

```cpp
#include <iostream>
#include <list>
using namespace std;

int main(){
  list<int> arr{4,52,23,6,9};
  list<int>::iterator it;
  for(it=arr.begin(); it!=arr.end(); it++)
    cout << *it << endl;
  return 0;
}
```

- Doubly-linked lists: bidirectional iteration, constant-time insertion/deletion at any point of the container
- Elements stored in non-contiguous memory, each at unrelated memory locations
- No fast random access, `list` must be traversed until the element is found

Associative containers

In associative containers, elements are stored and retrieved by a key. Two primary data structures `map` and `set` are tweaked in order to provide various functionality:

- Ordered
  `set`, `multiset`, `map`, `multimap`
- Unordered^C++11
  `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`

Associative containers handbook

- `map`
  Stores key-value pairs, with unique keys
- `set`
  Stores unique keys, with keys being also values
- `multi` containers
  Allow multiple copies of a key
- Unordered containers
  Key-value pairs are organized by a hash function rather than an ordering on keys, yielding faster individual element access

Using `map`

```cpp
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main(){
  map<string,string> phoneBook{{"A", "+39 389 2018312"}, {"B", "+1 202 555 0191"}};

  map<string,string>::iterator it;
  for(it = phoneBook.begin(); it != phoneBook.end(); it++){
    if(it->second == "+39 389 2018312"){
      cout << "Deleting tel. number +39 389 2018312 belonging to " << it->first << endl;
      phoneBook.erase(it);
      break;
    }
  }

  string person = "C";
  string number = "+86 16521687834";
  cout << "Inserting new tel. number for " << person << ": " << number << endl;
  phoneBook.insert(make_pair(person, number));

  string newNumber = "+39 06 432 9835";
  phoneBook["B"] = newNumber;

  return 0;
}
```

Using `set`

```
#include <iostream>
#include <set>
using namespace std;

int main(){
  int list1[5] = {1,2,3,4,5};
  int list2[5] = {3,4,5,6,7};
  set<int> unique(list1, list1 + 5);
  unique.insert(list2, list2 + 5);

  set<int>::iterator it;
  for(it = unique.begin(); it != unique.end(); it++)
    cout << *it << endl;

  return 0;
}
```

Range-based `for` loop

Starting from C++11, it is possible to use the for each loop by means of range-based `for`:

$$\texttt{for (}\underbrace{\texttt{range declaration}}\texttt{ : }\underbrace{\texttt{range expression}}\texttt{) statement}$$

Variable declaration, its type must be the same as elements of the sequence represented by `range expression`. Variables are often declared `auto`

Expression representing a sequence accepted by the range-based `for` loop: an array, object with `begin()` and `end()` functions, etc.

STL containers can be passed as `range expression` by variables' name without further effort, as the loop will use an iterator.
It must be noted that this loop acts on copies of the contents, therefore performance issues may arise if not carefully handled:

- `for (auto element : container) statement`
  Default behavior, elements are copied (capture-by-value)
- `for (const auto &element : container) statement`
  Capture-by-reference behavior, read-only; in case elements have to be modified, use a `auto &` variable declaration

4. More on functions

`const` pass by reference?

As previously mentioned, pass by reference is a method to provide arguments to functions in order to allow the values of the variables to be modified during the execution of a function and the new value to be exposed to the calling function.

```
void swapFloat(float &num1, float &num2);
```

What would happen if the function declaration included a `const float &` parameter instead?

The parameter would be passed in read-only mode, therefore the function would not be able to manipulate its value, although a reference was passed.

This is useful when dealing with memory-heavy parameters, where passing by value would take a long time due to the copy operation, but the function is not intended to access them in writing mode. Trying to do so will trigger a compile time error.

Array arguments for a function

C++ does not allow passing built-in array arguments to a function.

At least not directly

Arrays are implicitly converted to pointers despite the syntax might not show it. For instance:

```
int values[5] = {45, 7, 22, 980, 12};
int sumV = sumAll(values);
```

can be compiled with all the following `sumAll()` declarations, although the underlying implementation will always be the first one:

1. `int sumAll(int *array);`
2. `int sumAll(int array[10]);`
3. `int sumAll(int array[]);`

Declaration #2 does not provide any additional benefits (*e.g.* automatic size checking), and passing an argument specifying the array size is a good practice.

5. Exercises

Exercises

1. Write a program that, given two arrays of equal size, uses a function that merges them into a single array having elements from the first and the second one alternating every position
   (*e.g.* input: [1,2,3], [7,8,9], output: [1,7,2,8,3,9]; *hint:* function should accept three array parameters, the last one being double the size of the first two)
2. Implement the same program of Exercise 1 using STL containers (not necessarily of the same size) and iterators
   (*hint:* use `vector` and pass-by-reference mechanism)

References

- https://www.cplusplus.com/doc/tutorial/arrays/
- https://www.cplusplus.com/doc/tutorial/pointers/
- https://www.cplusplus.com/reference/stl/
- https://www.cplusplus.com/reference/iterator/
- https://en.cppreference.com/w/cpp/language/range-for