Advanced Topics in C/C++

Lecturer: Duc Dung Nguyen, PhD.
Contact: nddung@hcmut.edu.vn

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

Contents

Function Pointer

Function pointer

- Just a pointer storing the address of a particular function

Function pointer

- Just a pointer storing the address of a particular function
- Why do we need them?
  - Provide flexibility for your program
  - E.g.: Callbacks, Event-driven programs, etc.

Function pointer

- Just a pointer storing the address of a particular function
- Why do we need them?
    - Provide flexibility for your program
    - E.g.: Callbacks, Event-driven programs, etc.
- Syntax:

```
<return_type> (*funcName)(<parameter types>);
```

Function pointer

- Just a pointer storing the address of a particular function
- Why do we need them?
    - Provide flexibility for your program
    - E.g.: Callbacks, Event-driven programs, etc.
- Syntax:

```
<return_type> (*funcName)(<parameter types>);
```

- Example:

```
void (*funcPtr)(int);
// funcPtr is a variable that can point to any function of type void (int)
```

Function pointer

- Assign function pointer to a function:

```
float add(float a, float b) { return a + b; }
...
float (*op)(float, float) = add;
```

or

```
float (*op)(float, float);// op is a variable of type pointer
op = add;// op = &add
```

Function pointer

- Assign function pointer to a function:

```
float add(float a, float b) { return a + b; }
...
float (*op)(float, float) = add;
```

or

```
float (*op)(float, float);// op is a variable of type pointer
op = add;// op = &add
```

- Invoke the function:

```
op(1.5, 2.5);
(*op)(2.44, 3.1);
```

Function pointer

- Assign function pointer to a function:

```
float add(float a, float b) { return a + b; }
...
float (*op)(float, float) = add;
```

or

```
float (*op)(float, float);// op is a variable of type pointer
op = add;// op = &add
```

- Invoke the function:

```
op(1.5, 2.5);
(*op)(2.44, 3.1);
```

- Can we define a type of function pointer?

Function pointer

Facts

- A function pointer points to code rather than data
- We dont use function pointers to allocate or de-allocate memory
- The name of a function may also be used to find the address of that function
- Regular pointers can be used with an array of function pointers in the same manner that regular pointers can
- A function pointer **can be supplied as an argument** and **returned** from a function.

Callbacks

- A callback is a callable accepted by a class or function, used to customize the current logic depending on that callback.

Callbacks

- A callback is a callable accepted by a class or function, used to customize the current logic depending on that callback.
- Role of callback
  - To write generic code which is independent from the logic in the called function and can be reused with different callbacks.
  - Setting up "listener" or "callback" functions that are invoked when a particular event happens

```
void create_button(int x, int y, const char *text, function callback_func);
```

Callbacks

- A callback is a callable accepted by a class or function, used to customize the current logic depending on that callback.
- Role of callback
    - To write generic code which is independent from the logic in the called function and can be reused with different callbacks.
    - Setting up "listener" or "callback" functions that are invoked when a particular event happens

```
void create_button(int x, int y, const char *text, function callback_func);
```

- NOTE: The basic callback function in C++ does not guarantee asynchronous behavior in a program.

Lambda Expression[1]

- Lambda expression constructs a closure: an unnamed function object capable of capturing variables in scope.

[1] since C++11

Lambda Expression[1]

- Lambda expression constructs a closure: an unnamed function object capable of capturing variables in scope.
- Syntax:

```
[ captures ] ( params ) specs requires { body }
```

[1] since C++11

Lambda Expression[1]

- Lambda expression constructs a closure: an unnamed function object capable of capturing variables in scope.
- Syntax:

```
[ captures ] ( params ) specs requires { body }
```

- Example:

```
[] (float a, float b) -> float { return a + b; }
```

[1] since C++11

Smart Pointer

Smart Pointer

- Smart pointers are used to make sure that an object is deleted if it is no longer used (referenced)

Smart Pointer

- Smart pointers are used to make sure that an object is deleted if it is no longer used (referenced)
- Sounds similar? (you heard about it in Java)

```
#include <memory>
void my_func()
{
    std::unique_ptr<int> valuePtr(new int(15));
    int x = 45;
    // ...
    if (x == 45)
        return;    // no memory leak anymore!
    // ...
}
```

Shared Pointer

- The shared_pointer is a reference counting smart pointer that can be used to store and pass a reference beyond the scope of a function

Shared Pointer

- The shared_pointer is a reference counting smart pointer that can be used to store and pass a reference beyond the scope of a function
- Example:

```cpp
#include <memory>

class Foo {
    public void doSomething();
};
class Bar {
    std::shared_ptr<Foo> pFoo;
public:
    Bar() { pFoo = std::shared_ptr<Foo>(new Foo()); }
    std::shared_ptr<Foo> getFoo() { return pFoo; }
};
```

Regular Expression

Regex

- Look for a desired pattern in a string
- Example:

```cpp
#include <iostream>
#include <regex>
using namespace std;
int main() {
    regex reg("man");
    if (regex_search("Here is my man.", reg))
        cout << "matched" << endl;
    else
        cout << "not matched" << endl;
    return 0;
}
```

Pattern

- Regexes are usually complex
- Regular expressions have metacharacters
- Metacharacters are characters with special meanings. A metacharacter is a character about characters.

Pattern

- Regexes are usually complex
- Regular expressions have metacharacters
- Metacharacters are characters with special meanings. A metacharacter is a character about characters.
- C++ regex metacharacters are:

```
^ $ \ . * + ? ( ) [ ] { } |
```

Character Classes

- Square Brackets: a particular position in the target string would match any of the square brackets characters
- Some simple regexes: [cbr]at, [0-9], [a-z], [A-Z]

Character Classes

- Square Brackets: a particular position in the target string would match any of the square brackets characters
- Some simple regexes: [cbr]at, [0-9], [a-z], [A-Z]
- Example:

```
regex reg("[cbr]at");
if (regex_search("The cat is in the room.", reg))
    cout << "matched" << endl;
if (regex_search("The bat is in the room.", reg))
    cout << "matched" << endl;
if (regex_search("The rat is in the room.", reg))
    cout << "matched" << endl;
```

Character Classes

- Square Brackets: a particular position in the target string would match any of the square brackets characters
- Some simple regexes: [cbr]at, [0-9], [a-z], [A-Z]
- Example:

```
regex reg("[cbr]at");
if (regex_search("The cat is in the room.", reg))
    cout << "matched" << endl;
if (regex_search("The bat is in the room.", reg))
    cout << "matched" << endl;
if (regex_search("The rat is in the room.", reg))
    cout << "matched" << endl;
```

- Negation: [^0-9]

Character Classes

- Matching Whitespaces: ‘ ’ or \t or \r or \n or \f is a whitespace character.

```
if (regex_search("Of line one.\r\nOf line two.", regex("\n")))
    cout << "matched" << endl;
```

Character Classes

- Matching Whitespaces: ' ' or \t or \r or \n or \f is a whitespace character.

```
if (regex_search("Of line one.\r\nOf line two.", regex("\n")))
    cout << "matched" << endl;
```

- The period (.) in the Pattern: matches any character including itself, except \n, in the target

```
if (regex_search("1234abcd", regex(".")))
    cout << "matched" << endl;
```

Character Classes

- Matching Repetitions: The metacharacters, ?, *, +, and {} are used to match the repetition in the target
- x*: means match ’x’ 0 or more times, i.e., any number of times
- x+: means match ’x’ 1 or more times, i.e., at least once
- x? : means match ’x’ 0 or 1 time
- x{n,}: means match ’x’ at least n or more times. Note the comma.
- x{n} : match ’x’ exactly n times
- x{n,m}: match ’x’ at least n times, but not more than m times.

Character Classes

- Matching Alternation

```
char str[] = "The farm has pigs of different sizes.";
if (regex_search(str, regex("goat|rabbit|pig")))
    cout << "matched" << endl;
else
    cout << "not matched" << endl;
```

Character Classes

- Matching Alternation

```
char str[] = "The farm has pigs of different sizes.";
if (regex_search(str, regex("goat|rabbit|pig")))
    cout << "matched" << endl;
else
    cout << "not matched" << endl;
```

- Matching Beginning or End

```
if (regex_search("abc and def", regex("^abc")))
    cout << "matched" << endl;
```

```
if (regex_search("uvw and xyz", regex("xyz$")))
    cout << "matched" << endl;
```

Remarks

- The regular expression uses patterns to match substrings in the target sequence string.
- Patterns have metacharacters. Commonly used functions for C++ regular expressions, are: regex_search(), regex_match() and regex_replace().
- A regex is a pattern in double-quotes.
- The regex must be made into a regex object before these functions can use it.

lvalue vs. rvalue

lvalue vs. rvalue

- **lvalue** is something that points to a specific memory location
- **rvalue** is something that doesn’t point anywhere

lvalue vs. rvalue

- **lvalue** is something that points to a specific memory location
- **rvalue** is something that doesn't point anywhere
- Example

```
int x = 666;
```

lvalue vs. rvalue

- **lvalue** is something that points to a specific memory location
- **rvalue** is something that doesn't point anywhere
- Example

```
int x = 666;
```

- *You are allowed to bind a const lvalue to an rvalue*

Unit Test

Google Unit Test (GTest)

- There are some frameworks for unit testing, and GTest is one of them.
- Google C++ Testing is based on xUnit architecture. It is a cross platform system that provides automatic test discovery.
- It supports a rich set of assertions such as fatal assertions (ASSERT_), non-fatal assertions (EXPECT_), and death test which checks that a program terminates expectedly.

- Please refer to the technical document.

Google Unit Test (GTest)

- There are some frameworks for unit testing, and GTest is one of them.
- Google C++ Testing is based on xUnit architecture. It is a cross platform system that provides automatic test discovery.
- It supports a rich set of assertions such as fatal assertions (ASSERT_), non-fatal assertions (EXPECT_), and death test which checks that a program terminates expectedly.
- Example

```
#include "sample1.h"
#include <limits.h>
#include "gtest/gtest.h"
namespace { // Tests factorial of negative numbers.
TEST(FactorialTest, Negative) {
  EXPECT_EQ(1, Factorial(-5));
  EXPECT_EQ(1, Factorial(-1));
  EXPECT_GT(Factorial(-10), 0);
}}
```

- Please refer to the technical document.