Basic OOP in C++

Overview

- Input Files
- Dynamic Arrays
- Structs
- Type Aliases (formally typedefs)
- Functional types
- Lambda Expressions

References

- Gary J. Bronson: C++ for Engineers and Scientists. 3rd Edition. Thomson (2010)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)

Opening an Input File

- We can create an input file stream object and attach an input file in one step:

```
ifstream lInput( aFileName );
```

- If aFileName denotes a path to the input file, then this declaration instantiates object lInput – a file input stream – and the constructor will call the open method using text mode – ifstream::in.

Referring to this object

- We often have to refer to "this object" inside a method. However, the implicit first parameter **this** is a pointer that is initialized with the address of the object on which the method was invoked.
- We can use

```
*this
```

to access "this object." In other words, we have to dereference the pointer **this** to obtain the object that is located at the address denoted by the value of **this**.

Dynamic Arrays

- We regularly need dynamically-sized arrays.
- These array can and have to be created on the heap using the new expression:

```
new type[n];
```

The array holds n elements of the indicated type. The operator new returns a pointer to the first element in the array.

Releasing Memory of Dynamic Arrays

- Every time we allocate a dynamically-sized array, we eventually have to **delete** it. That is, we have to release its memory.
- We can use the **delete** expression for this purpose:

delete pointer;

★ **delete** [] pointer;

The operator **delete** expects a pointer. This pointer has to denote valid address of a previously allocated heap object, or it can be **nullptr**. If pointer refers to an array, then we need to use **delete** [] pointer. This expression performs two operations: first, it frees all the objects in the array and, second, it frees the space associated with the array. The expression **delete** pointer just releases the memory associated with pointer, and if pointer refers to an array that contains other heap-based objects, potentially creates memory leaks.

Structures

- We can create structures in C++ which are classes that have implicit public access initially:

```
struct AStruct
{
    field-or-function-declarations
};
```

The **struct** keyword is inherited from C. Structs in C++ are class types with default access level public.

- When defining structs we can omit the name declaration. This results in an anonymous struct, which is often used in a context where we require an alternative view on data (e.g., tagged unions).

Using Declaration — C++11 Type Aliases

- A type alias is a name that refers to a previously defined type.

```
using identifier = type;
```

- Type aliases are commonly used for three purposes:
  - To hide the implementation of a given type.
  - To streamline complex type definitions making them easier to understand, and
  - To allow a single type to be used in different contexts under different names.
- Type aliases establish a nominal equivalence between types.
- Type aliases are similar to **typedef**. However, type aliases are better suited when creating alias templates.

Using & Struct

```
using DataMap =
    struct
    {
        size_t fIndex;
        size_t fDatum;

        const char getAsChar() const;
    };
```

same

```
struct DataMap
{
    size_t fIndex;
    size_t fDatum;

    const char getAsChar() const;
};
```

std::function<class Ret, class... Args>

- Technically, a variable that stores a lambda is a function pointer. However, function pointers may lead to unreadable specifications or worse.
- C++11 offer a function wrapper std::function<class Ret, class... Args> for this purpose.
- Technically, std::function is a varadic template (it takes a variable number of arguments) that allows us to capture any function signature.
- For example:

```
using StringMap = std::function<size_t(const std::string&)>;
```

defines type StringMap as a function from const string& to size_t using C++11's typedef declaration (i.e, using TypeName = aType;).

Callable

use functional templates

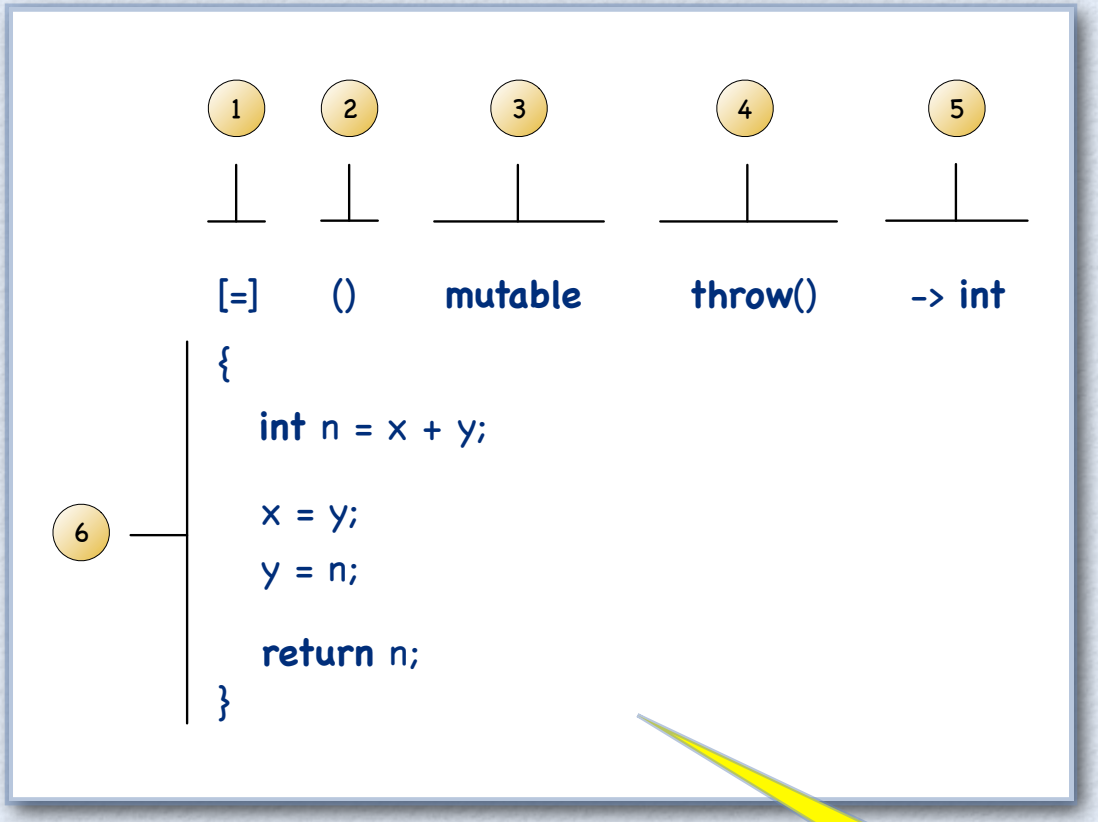
```
#include <functional>


using Callable = std::function<const char(size_t)>;
```

type alias

function from size_t to const char

C++ Lambda



1. Capture clause
2. Parameter list, optional
3. Mutable specification, optional
4. Exception specification, optional
5. Trailing return type, optional
6. Lambda body

Variables x and y are captured by value, but can be altered within the body of lambda.

Lambda Capture List

[]	Lambda does not use variables from the enclosing environment. Variables from the environment cannot be accessed.
[identifier list]	The variables listed in the comma-separated identifier list are captured by value and copied into the body of lambda. The lambda sees only stored values. Updates in the environment have not effect on lambda.
[&]	All variables in the environment are implicitly captured by reference. Updates in the environment affect lambda.
[=]	All variables in the environment are implicitly captured by value. Values are copied into the body of lambda. Updates in the environment have no effect on lambda.
[&, identifier list]	Implicit capture by reference of all variables in the environment, except those that occur in identifier list. Identifier list must not contain &.
[=, reference list]	Implicit capture by value (copied into the body of lambda) of all variables in the environment, except those that occur in identifier list. Reference list may not contain **this** and all names must be preceded by &.

Lambda lIdentityMapper

use auto type

capture lData by reference

throw declaration

```
auto lIdentityMapper = [&lData] (size_t aIndex) throw(out_of_range) -> const char
{
    if ( aIndex < lData.size() )
    {
        return lData[aIndex].getAsChar();
    }

    throw out_of_range( "Invalid index." );
};
```

result type

function from size_t to const char

Easter Eggs in Software

- In computer software, Easter eggs are secret responses that occur as a result of an undocumented set of commands.
- An Easter Egg is a purposely hidden message (or joke) in the object code or data used by the program.