Akademin

Functional Programming & Modularity and Data Structures

C++ Language

Functional Programming and Modularity

- **Functional programming** is a programming paradigm where programs are constructed by applying and composing functions.
  - Functions are the main building blocks of a program.
- A module is a self-contained part of a system that has a **well-defined interface** and an **implementation** to hide the details in order to provide some functionalities for the system
- Modularity helps us to
  - Organize large programs in smaller parts (modules)
  - Make testable, understandable and reusable code
  - Provide clear and flexible interfaces
  - Have separate compilation and make changes easier
  - Hide implementation and support operations on data structure
  - Create abstract data type and hide details

Akademin

Modularity and Functional Programming

- C++ supports modularity using both functional and object oriented programming.
- Generally there are two types of module
  - Single instance modules which have a single set of data to manage
  - Multiple instance modules which shall be able to manage multiple sets of data
- The smallest possible module consists of a **header file** and a **source file**.
  - The **header file** contains the module's public interface.
    - According to encapsulation principles it shall only reveal what which is required by the module clients. Even if a module does not manage any data it shall follow this principle. Generally encapsulation protects objects from unwanted access by clients.
    - There are cases in which a module is can/shall be implemented in only a header file.
  - The **source file** contains the implementation of the public interface in order to hide the details of the module and restrict access to identifiers out of the source file.

Modularity

- To make a module we shall follow some rules and conventions
  - Choose a unique name for the module
  - Make a header file and a source file with the same name as the module name.
    - All the accessible identifiers in the header file shall start with name of the module
      - It also is possible to use namespaces
    - Always we shall have an inclusion guard for a header file.
    - Declare functionalities and all the client of the module needs in the header file.
    - The header file and identifiers in the header file shall be well-described.
      - The header file is the public interface and descriptions are like the module manual.
    - Always include the header file in the source file.
  - Ensure that clients of the module have access only to those identifiers which are required to use the functionalities provided by the module; not more, not less.

C++ and Functional Programming

- Only one definition of any variable, function, class type, enumeration type, or template is allowed in any one translation unit. Definitions are declarations that fully define the entity introduced by the declaration. There are exceptions. Look at One Definition Rule
- If in the scope of a **source file** we have a:
  - A static variable or function, its scope is limited to the file and out of the file is not accessible.
    - Note that the default storage class of a function is extern and globally it is accessible.
  - A type or macro, its scope is limited to the file.
  - A const variable, its scope by default is the file.
    - But it is possible to make it extern and using extern get access to it out of the file.
  - A constexpr variable, its scope is limited to the file.
    - Impossible to get access to it out of the file.
  - An inline function, its definition must be reachable. An inline function may be defined in a header

C++ and Functional Programming

- If an identifier in another header is used only in the source file of the module, the header file shall be included only in the source file. In this way we can make a module private in another module. E.g. only implementation of module A depends on module B; header file of B shall be included only in the source file of A.
- Generally we shall not have program scope global variables and therefore we shall not define non-constexpr objects in a header file.
- constexpr functions are functions that may evaluated at compile-time. E.g. `constexpr int add(int x, int y) { return (x + y); }`
  - constexpr functions are implicitly inline functions
  - constexpr functions can also be evaluated at runtime
- In C++ 17 it also is possible to have inline variables

```cpp
#include <iostream>
constexpr int add(int x, int y) { return (x + y); }
int main(void) {
    constexpr double gravity{9.8}; // Compile-time const
    int id{};
    std::cout << "Enter your age: ";
    std::cin >> id;
    const int my_id{id}; // A runtime constant
    constexpr int a{add(20, 30)};

    // Error: value of id is unknown during compilation
    constexpr int your_id{add(10, id)};
    return 0;
}
```

Akademin

Modularity

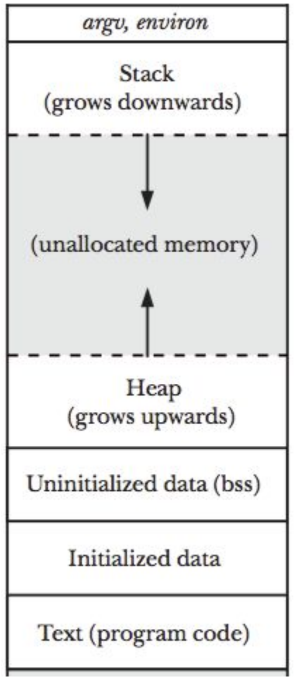
- In a **single instance** module we can define variables and function as static to make them them private. If it is required, make the private data accessible through the module's public interface. By defining a set of function prototypes in the header file.
- In **multiple instance** module to hide the data details we shall create incomplete types to abstract data. To create an incomplete type we shall:
  - Have the forward declaration of the type in the header file. E.g. `struct Person;`
  - Define the type in the source file. E.g. `struct Person { int id; int age; }`
  - We can not create instances of Person out of the source file.
    - We can only create pointers to instances of Person and a pointer to an incomplete type cannot be dereferenced.
  - To create instances during run time we shall use dynamic memory allocation

Modularity

- When we allocate memory dynamically we are responsible to release it.
- Usually in a **single instance** module we have an initialization and a deinitialization functions
- In **multiple instance** module we shall have a create and and a destroy functions
  - The create function is used to create instances using dynamic memory allocation
  - The destroy function is used to release the dynamically allocated memory for an instance
- **Dynamic Memory Management**
  - In this mechanism we can allocate and deallocate memory dynamically during runtime
  - The heap or free store is a large pool of memory; it can be used for storing dynamic size data
  - The programmer is responsible to manage the dynamic memory
  - Data stored in the heap, is alive until we release the allocated memory for the data
    - Unlike local variables which exist only in their scops

Dynamic Memory Management

- What we know about different segments of a program
  - **Code Segment** is used to store the program code.
  - The global and static variables and constant string literals are stored in the **Data Segment**.
  - A **Stack** is used to store local variables in functions and
  - and the required data for functions calls and returns
  - Heap is used to for dynamic objects during run time
- Sometimes size of data a program should process is unknown during compilation and it will be specified during run time. E.g. When we ask a user to enter her name
  - Size of the name is unknown during compilation



Memory Layout of a C++ Program

Environment is the used for the environment variables and arguments passed to the program. **BSS** (Block Started by Symbol) includes the uninitialized global and static variables. **DS** includes the global initialized constants , static variables and string literals.

Akademin

Dynamic Memory Management

❖ In C++ dynamic memory management can be done in two ways

- ➢ The C functions defined in **<cstdlib>**:
  - void *malloc(size_t **size**);
  - void *calloc(size_t **count**, size_t **size**);
  - void *realloc(void *ptr, size_t size);
  - void free(void *ptr);
- ➢ The new and delete operators of C++
- ➢ We shall not mix them. E.g. using malloc to allocate memory and free it using delete
- ➢ ***As the best practice; use only new and delete***

❖ ***When we allocate memory, always we have to check for failure***

❖ There are some useful functions in **<cstring>** to operate on blocks of memory

- ➢ E.g. memset, memcmp, memcpy, memmove and etc.

Dynamic Memory Management

- In C++, the new keyword is used to allocate memory dynamically. E.g. int* ptr{ new int };
- It is possible to initialize the allocated memory:
  - int *ptr1{new int(5)}; int *ptr2{new int{6}};
- To check if a memory allocation is successful or not

```
int *ptr{new (std::nothrow) int{}}; // ask for memory allocation
if (ptr == nullptr) // handle the case where new returned null
{
  // Do error handling here
}
```

  - If new fails, a std::bad_alloc exception is thrown and we can catch and handle it.
  - Alternatively we can add the std::nothrow constant between the new keyword and the allocation type. In this case if new fails a nullptr is return.
- To free an allocated memory, you can use the delete keyword. (used only for dynamic objects)
  - delete ptr; ptr = nullptr;   // Release the allocated memory and set ptr to nullptr as the best practice
- Memory allocation and deallocation for arrays
  - int *array{new int[5]{9, 7, 5, 3, 1}}; // Allocate memory
  - delete[] array; array = nullptr; // Free the allocated memory

C++ Assertion

- An assertion is a boolean expression that will be true unless there is an issue in the program.
- Dynamic assertion
  - Assertions in runtime
  - Include <cassert> and use the assert macro
- Static assertion
  - Assertions in compile time
  - Use the static_assert keyword

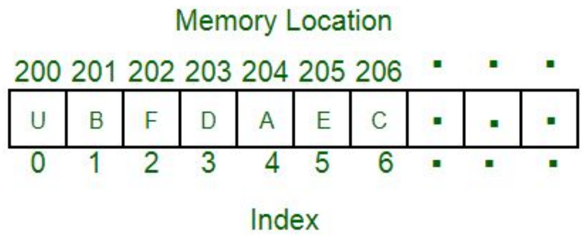
    static_assert(condition, diagnostic_message)

    - condition must be evaluable at compile time
  - A failing static_assert causes a compilation error

```
#include <cassert>
#include <iostream>

static_assert(sizeof(long) == 8, "long must be 8 bytes");
static_assert(sizeof(int) == 4, "int must be 4 bytes");

void display_number(int *iptr) {
  assert(iptr != nullptr && "iptr shall not be nullptr");
  std::cout << "iptr contains value"
        << " = " << *iptr << std::endl;
}

int main(void) {
  int var = 5;
  int *ptr1 = &var;
  int *ptr2 = nullptr;

  display_number(ptr1);
  display_number(ptr2);
  return 0;
}
```

Data Structure - Array

- Data structure: The way of organizing data for particular types of operation
- Some common data structures:
  - Array
  - Linked List
  - Queue
    - Circular Queue
  - Stack
  - Tree
  - And etc.

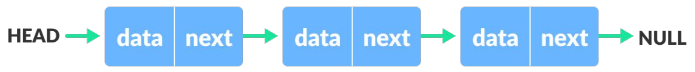


Array is a **fixed size** collection of items stored consecutively in a continuous piece of memory. Arrays allow random access of elements. Each element in an array can be accessed by its position in the array which is the **index** of the element.
The index of the first element is 0 and the index of the last element is the number of the elements - 1.
Arrays have a better **cache locality** that can make a big difference in **performance**.

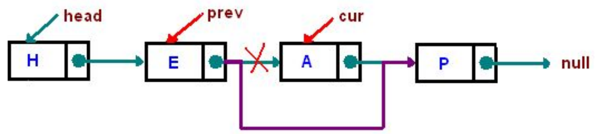
Akademin

Data Structure - Linked List

- A linked list is a way to make a **dynamic list** as a series of connected nodes using pointers.

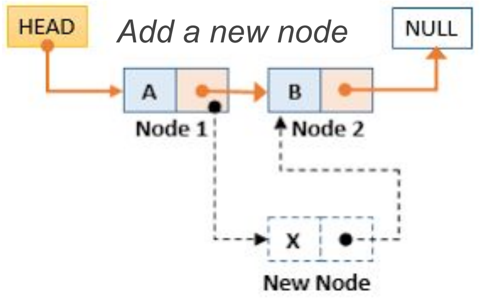


- The most common type is a **singly linked list**
  - Each node points to the next node
- We can dynamically add a new node
- We can delete any node in the list
- It does not allow random access to the nodes. To access a specific node we need to traverse the linked list usually starting from the **head** pointer which points to the first element.
- In a **doubly linked list** there is also a pointer to the previous node
  - We can traverse the list in the forward and backward directions



Delete a node (A) from a linked list

```
struct node_t {
    int data;
    node_t *next;
};
```

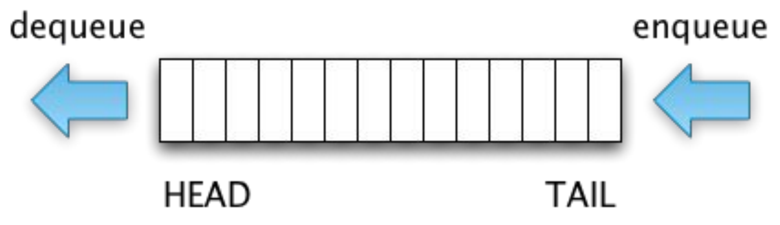


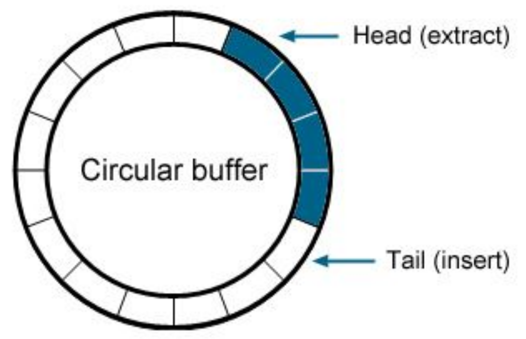
```
struct node_t {
    int data;
    node_t *next;
    node_t *previous;
};
```

Akademin

Data Structure - Queue

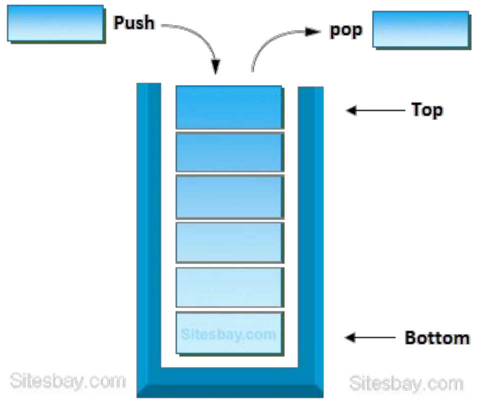
- ❖ A queue is a linear data structure to store and retrieve data elements.
- ❖ It follows the order of First In First Out (FIFO).
- ❖ In a queue, the first entered element is the first one to be removed from the queue.
- ❖ Typical Operations Associated with a Queue
  - ➢ isempty(): To check if the queue is empty or not
  - ➢ isfull(): To check whether the queue is full or not
  - ➢ dequeue(): Removes the element from the frontal side of the queue
  - ➢ enqueue(): It inserts elements to the end of the queue
- ❖ A queue can be implemented using
  - ➢ A fixed size array - It is called ring or circular queue/buffer.
  - ➢ A dynamic size array or even using a linked list.





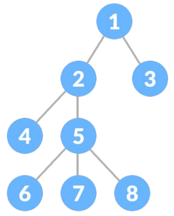
Data Structure - Stack

- A stack is a linear data structure to store and retrieve the data elements.
- It follows the order of Last In First Out (LIFO).
- In a stack, the first entered element is the last one to be retrieved from the stack.
- Typical Operations Associated with a Stack
  - isempty(): To check if the stack is empty or not
  - isfull(): To check if the stack is full or not
  - pop(): Removes an item from the top of the stack.
    - If the stack is empty we have an underflow condition
  - push(): Inserts an item in the stack.
    - If the stack is full we have an overflow condition.
- A stack can be implemented using an array (fixed or dynamic size) or a linked list

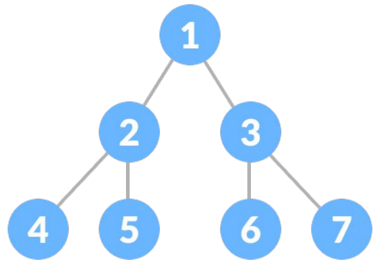


Akademin

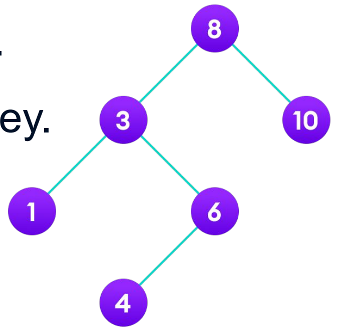
Data Structure - Binary Tree



- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges (pointers)
- A binary tree is a tree data structure in which each parent node can have at most two children.
- Binary Search Tree (**BST**) is a binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys lesser than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - The left and right subtrees each must also be a binary search tree.
- Typical operations on a BST: insert, edit, delete, search and traverse
- A BST is automatically sorted. It provides quicker access/search than linked lists



```
struct node_t {
  int data;
  node_t *left;
  node_t *right;
};
```



C++ Language

- Some useful links
  - C++ Reference
  - Standard C++ Library Reference
  - C++ Tutorial
  - C++ Language
  - C++ Tutorial
  - C++ Full Course For Beginners
  - C++ Tutorial for Beginners - Full Course
  - C++ Tutorial 2021
  - C++ Programming Tutorials Playlist
  - Linked List Data Structure
  - Stack Data Structure
  - Queue Data Structure
  - Circular Buffer Structure
  - Ring Buffer (Circular Buffer)
  - Binary Tree Data Structure
  - Binary Search Tree (BST)
  - Data Structures Easy to Advanced Course