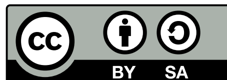
Common Data Structures for Collections & Multi-File Programs

The Art of Computer Programming

Pierre Senellart



3 November 2025

Outline

Common Data Structures for Collections

Collections

Ordered Sequences

Value-Based Collections & Associative Arrays

Multi-File Programs

Reminder: What are Collections?

- In computer science, a collection is a group of elements stored together and accessible through a unified interface
- Collections allow us to:
  - Store multiple values
  - Organize them logically
  - Access, modify, and process them efficiently
- Different kinds of collections support different forms of operations and are used for different purposes

Reminder: Abstract Kinds of Collections

Ordered sequences Elements are arranged in order; two subkinds:

- With random access, i.e., possibility of directly accessing any element of the sequence (arrays)
- Without random access, i.e., only possible to directly access specific elements, such as the first or the last

Value-based collections (e.g., sets), where there is no underlying sequential order of the elements, but we only care about whether an element is present in the collection

Associative arrays or maps or key–value stores: each element is formed of a pair of a key and a value, access to elements is by their key, duplicate keys are (usually) not allowed

Outline

Common Data Structures for Collections

Collections

Ordered Sequences

Value-Based Collections & Associative Arrays

Multi-File Programs

Ordered Sequences

- Data structures that store a sequence $S = (s_0, \ldots, s_{n-1})$ of n elements (integers, floating-point numbers, complex objects, etc.)
- Different specifications of such structures allow for different operations, with varying efficiency:
  - Random access: given i, retrieve s_i
  - Access to the beginning (s_0), or to the end (s_{n-1})
  - Insertion at the beginning (before s_0), at a random position (between s_i and s_{i+1}), or at the end (after s_{n-1})
  - Deletion of the first element (s_0), of a random element (s_i), or of the last element (s_n)
  - Search for an occurrence of an element s (return an i such that $s_i = s$); always in $\Theta(n)$ for the structures described here

Fixed-size array

- Contiguous memory area of fixed size, allocated upon creation
- Random access in $\Theta(1)$: if the array starts at address t in memory and stores elements of k bytes each, it suffices to access the element at address $t + ki$
- Insertion and deletion impossible
- As compact as possible – no wasted space

Fixed-size arrays in Python

- Do not exist in the standard library (though Python lists, implementing dynamic arrays can be used)
- `numpy.array` in the third-party (but widely used) package numpy
  - Stores homogeneous elements: primitive types (for example, integers between -2^{31} and $2^{31}-1$; 64-bit floating-point numbers; etc.) or more complex types built from primitive ones
  - Many features particularly suited to the storage and efficient processing of numerical values
  - Special case of a `numpy.ndarray`: a multi-dimensional array

Fixed-size arrays in C and C++

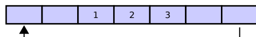
- C arrays (e.g., declared with `T array[42];`) are fixed-size arrays (but be careful of their use, e.g., decaying to pointers when passed to functions) – usable in C++ as well
- C++ also has the `std::array<T>` type; somewhat easier to use than C-style arrays, handled like other C++ collections
- C++ dynamic arrays (`std::vector<T>`) may also be used in settings when one would use fixed-size arrays – little disadvantages w.r.t. fixed-size arrays except for slightly larger memory use

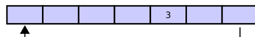
Dynamic array

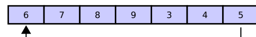
- Contiguous memory area, but with a variable logical size and a fixed physical capacity
- When the array becomes full and a new element is inserted, a larger block of memory is allocated (usually with twice the capacity), and the existing elements are copied over
- Amortized insertion at the end in $\Theta(1)$; worst case $O(n)$ when resizing occurs
- Random access in $\Theta(1)$, as for fixed-size arrays
- Deletion at the end in amortized $\Theta(1)$; deletion or insertion in the middle in $O(n)$
- Slightly less compact than a fixed-size array, since some extra space is reserved to avoid frequent reallocations
- In Python: Python lists are (heterogeneous) dynamic arrays; `array.array` from the Python standard library are homogeneous dynamic arrays
- In C++: `std::vector<T>` are homogeneous dynamic arrays

Double-ended dynamic array

- Goal: achieve amortized $\Theta(1)$ complexity for adding and removing elements at both the beginning and the end of an array, while keeping random access in $\Theta(1)$
- Two possible implementations:
  - A dynamic array filled from the middle – when the beginning or end of the allocated array is reached, a larger array is created
  - A dynamic array implementing a circular buffer – same concept, but if one end of the array is reached and there is space at the other end, elements are placed there until the array is fully filled

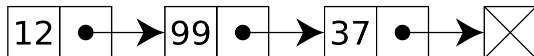






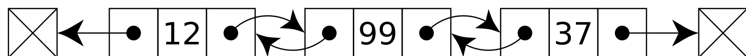
- No double-ended dynamic array in Python's standard library
- No double-ended dynamic array in the C++ STL; but see third-party `boost::circular_buffer`
- Conceptually close to the double-ended queue, see further

Singly linked list



- Each element is stored in a node, which also contains a pointer to the next element
- A pointer to the first node is also kept
- Access to the first element in $\Theta(1)$
- Random access or access to the last element in $O(n)$
- Insertion at a random position (after random access) in $\Theta(1)$
- Deletion of the first element in $\Theta(1)$
- Deletion in $\Theta(1)$ if the previous element is known, otherwise in $O(n)$

Doubly linked list



- Each element is stored in a node, which also contains a pointer to the previous and next elements
- A pointer to the first and last nodes is also kept
- Access to the first or last element in $\Theta(1)$
- Random access in $O(n)$
- Insertion at a random position (after random access) in $\Theta(1)$
- Deletion of the first or last element in $\Theta(1)$
- Deletion (after random access) in $\Theta(1)$
- More powerful, but less compact than the singly linked list

Linked lists in Python

- No singly linked list in the standard library
- `collections.deque` is a variant of a doubly linked list in which each node stores not one element but a fixed number of elements (currently 64 in the reference implementation)
- Mainly used to implement double-ended queues (see later)

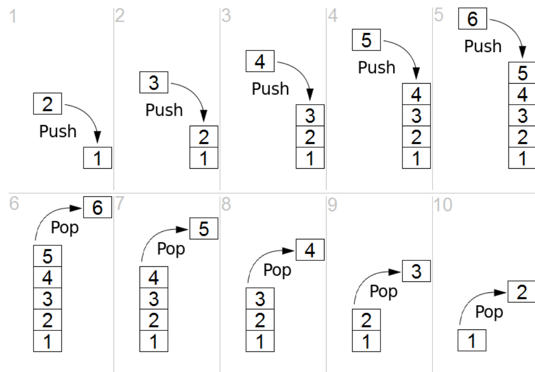
Linked list in C++

- Standard library provides `std::forward_list<T>` (singly linked) and `std::list<T>` (doubly linked)
- Use `.front()` and `.back()` to access the front or back elements
- Use `.push_front()` and `.push_back()` to add an element at the front or back
- Use `.pop_front()` and `.pop_back()` to remove an element at the front or back

C++

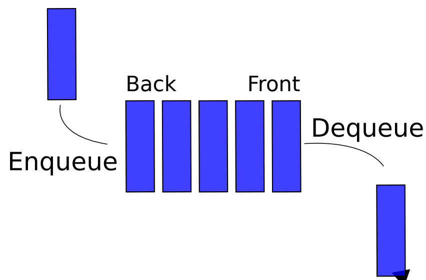
```cpp
#include <list>
#include <iostream>
using namespace std;

int main() {
  list<int> l = {1, 2, 3};
  l.push_front(0);          // O(1)
  l.push_back(4);           // O(1)
  l.pop_front();            // O(1)
  return 0;
}
```

Stack (*LIFO*: last-in-first-out)



- Abstract data structure, which can be implemented in various ways, e.g., with a singly linked list or a dynamic array
- Only possible operations:
  - Access the first element in $\Theta(1)$
  - Insertion at the beginning in $\Theta(1)$ (push)
  - Deletion of the first element in $\Theta(1)$ (pop)

Queue (*FIFO*: first-in-first-out)



- Abstract data structure, which can be implemented in various ways, e.g., using a doubly linked list or a double-ended dynamic array
- Only possible operations:
  - Access the first element in $\Theta(1)$
  - Insertion at the end in $\Theta(1)$ (enqueue)
  - Deletion of the first element in $\Theta(1)$ (dequeue)

Double-ended queue (deque)

- Abstract data structure, which can be implemented in various ways, e.g., using a doubly linked list or a double-ended dynamic array
- Combines the operations of stacks and queues
- Only possible operations:
  - Access the first or last element in $\Theta(1)$
  - Insertion at the beginning or at the end in $\Theta(1)$
  - Deletion of the first or last element in $\Theta(1)$
- Sometimes useful variant: double-ended queue with a maximum size – when full, adding a new element removes one from the opposite end

Stacks and queues in Python

- No simple stack or queue in the standard library
- But collections.deque implements a double-ended queue with a doubly linked list (with several elements per node)
- Operations: `append`, `appendleft`, `pop`, `popleft` – for instance, one can use append and pop for a stack, and `appendleft` and pop for a queue
- A `maxlen` parameter can be specified at creation to define a maximum size – by default, there is none
- For stacks, one can also use dynamic arrays
- There exist `queue.LifoQueue` and `queue.Queue`, but they are mainly intended for concurrent programming

Stacks and queues in C++

- The standard library provides `std::stack<T>` (LIFO) and `std::queue<T>` (queue), `std::deque<T>` (double-ended queue)
- `std::stack` and `std::queue` are adapters – they can use a `std::vector`, `std::list`, or `std::deque` (by default) as underlying storage: for example `std::stack<int, std::list<int>>` is a stack of `int` implemented using a `std::list<int>`
- Operations:
  - push / pop for `std::stack`
  - push_back / pop_front for `std::queue`
  - push_back, push_front, pop_back, pop_front for `std::deque`
- `std::deque` may be implemented in different ways, but provides the $\Theta(1)$ guarantees for front/back access, insertion, removal

Comparison of Ordered Sequence Data Structures

	Access			Insertion			Deletion		
Data structure	R	F	B	R	F	B	R	F	B
Fixed-size array	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	—	—	—	—	—	—
Dynamic array	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$\Theta(1)^*$	$O(n)$	$O(n)$	$\Theta(1)^*$
DE dyn. array	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$\Theta(1)^*$	$\Theta(1)^*$	$O(n)$	$\Theta(1)^*$	$\Theta(1)^*$
Singly linked list	$O(n)$	$\Theta(1)$	$O(n)$	$O(n)^\dagger$	$\Theta(1)$	$O(n)^\dagger$	$O(n)^\dagger$	$\Theta(1)$	$O(n)^\dagger$
Doubly linked list	$O(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)^\dagger$	$\Theta(1)$	$\Theta(1)$	$O(n)^\dagger$	$\Theta(1)$	$\Theta(1)$
Stack (LIFO)	—	$\Theta(1)$	—	—	$\Theta(1)$	—	—	$\Theta(1)$	—
Queue (FIFO)	—	$\Theta(1)$	—	—	—	$\Theta(1)$	—	$\Theta(1)$	—
DE queue	—	$\Theta(1)$	$\Theta(1)$	—	$\Theta(1)$	$\Theta(1)$	—	$\Theta(1)$	$\Theta(1)$

R Random (arbitrary position)

F First position

L Last position

$*$ amortized

$\dagger$ or $\Theta(1)$ if we already have a pointer

Outline

Common Data Structures for Collections
- Collections
- Ordered Sequences
- Value-Based Collections & Associative Arrays

Multi-File Programs

Beyond Ordered Sequences: Value-Based Collections and Maps

- Ordered sequences (arrays, lists) have limitations:
  - Assume a fixed order of elements
  - No efficient filtering of duplicates
  - Indexing elements by contiguous integers only
  - Searching for a specific element requires $\Theta(n)$
  - Insertion or deletion at arbitrary positions requires $\Theta(n)$

Beyond Ordered Sequences: Value-Based Collections and Maps

- Ordered sequences (arrays, lists) have limitations:
  - Assume a fixed order of elements
  - No efficient filtering of duplicates
  - Indexing elements by contiguous integers only
  - Searching for a specific element requires $\Theta(n)$
  - Insertion or deletion at arbitrary positions requires $\Theta(n)$
- We are looking for data structures with:
  - Potentially no underlying order of elements
  - Possibility of filtering duplicates
  - Possibility of indexing elements by an arbitrary key
  - Efficient search
  - Efficient insertion and deletion of arbitrary elements

Beyond Ordered Sequences: Value-Based Collections and Maps

- Ordered sequences (arrays, lists) have limitations:
    - Assume a fixed order of elements
    - No efficient filtering of duplicates
    - Indexing elements by contiguous integers only
    - Searching for a specific element requires $\Theta(n)$
    - Insertion or deletion at arbitrary positions requires $\Theta(n)$
- We are looking for data structures with:
    - Potentially no underlying order of elements
    - Possibility of filtering duplicates
    - Possibility of indexing elements by an arbitrary key
    - Efficient search
    - Efficient insertion and deletion of arbitrary elements
- Two variants:
    - Value-based collections (sets, multisets) – no indexing
    - Associative arrays (maps, key-value stores) – indexing of elements by an arbitrary key

Elements vs Key-Value Pairs

- Value-based collections and associative arrays are very similar:
  - Collection of elements (without any underlying sequential structure)
  - Collection of keys (without any underlying sequential structure), and for each key an associated value
- We can use the same techniques for both: instead of storing simple elements, store key–value pairs
- Only the key of a key–value pair is used to determine the identity of the stored element, to access the element – but the value is retrieved along the key

Implementations of Value-Based Collections

- Two typical implementations:
  - as balanced search trees
  - as hash tables

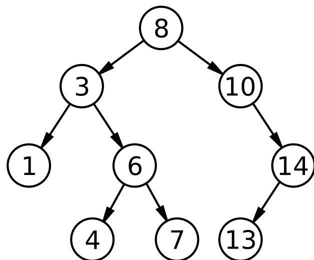
Implementations of Value-Based Collections

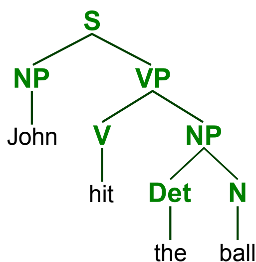
- Two typical implementations:
  - as balanced search trees
  - as hash tables
- The design of both is fairly complex, we just give a high-level overview

Implementations of Value-Based Collections

- Two typical implementations:
  - as balanced search trees
  - as hash tables
- The design of both is fairly complex, we just give a high-level overview
- True for data structures used for the design of algorithms and implemented in programming languages, but also in other settings:
  - for data stored on disk, especially within database management systems (e.g., B+-trees, hash files)
  - in distributed computing, for storing large collections of data (e.g., the domain name system (DNS) is a distributed search tree; peer-to-peer systems such as BitTorrent uses distributed hash tables)

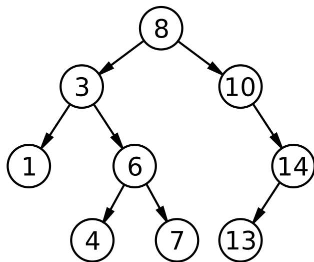
Binary Trees





- A binary tree is a hierarchical data structure:
  - Collection of nodes
  - Each node stores a value and has up to two children (left and right); it is said to be the parent of these two nodes
  - The top node is called the root
  - Nodes with no children are called leaves
- Binary trees are very common in computer science:
  - Naturally tree-like data, such as the parsing of a mathematical expression or of natural language
  - Foundation for other data structures, such as balanced binary search trees
- Depth of a node: length of the path from the root to that node (the root has depth 0, its children 1, etc.)
- Height of a tree: maximal depth of a node

Balanced Binary Search Trees (BST)



- A BST is a binary tree such that, for each node n:
  - all values in the subtree whose root is the left child of n are $\leqslant n$;
  - all values in the subtree whose root is the right child of n are $> n$.
- Operations:
  - Search for a value, from root to leaves
  - Insertion of a new value
  - Deletion of a value
  - Ordered traversal of the tree (from the smallest element to the largest)
- Insertion, Deletion: sophisticated algorithms (e.g., red–black trees, AVL trees) to ensure that the tree is self-balancing, i.e., that its height is $\Theta(\log n)$
- Complexity (when self-balancing): $O(\log n)$ for search, insertion, deletion

BST in Python

- No built-in BST in standard Python library
- Recommended third-party package: `sortedcontainers`

BST in C++

In the C++ STL, `std::set` and `std::map` are implemented as balanced BSTs

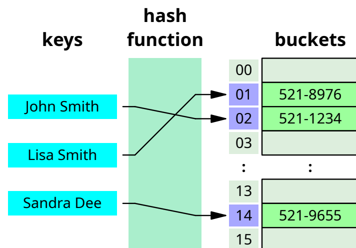
C++

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

int main() {
  set<int> bst;
  bst.insert(10);
  bst.insert(5);
  bst.insert(15);
  if(bst.find(10) != bst.end())
    cout << "Found 10\n";
  return 0;
}
```

Basics of Hashing

- Hashing maps data to a fixed-size value (hash code)
- A hash function $h(k)$ takes a key k and produces an integer index of small size (e.g., an integer between 0 and $2^m - 1$)
- Goal: efficient lookup, insertion, and deletion of elements
- Collisions may occur (two keys mapping to the same integer) but can be rare if h is a good hash function and m is large enough

Hash Tables (Principles)



- A hash table is an array of buckets storing elements (or key–value pairs for associative arrays)
- Operations:
  - Insert: compute hash, add element
  - Search: compute hash, check bucket
  - Delete: compute hash, remove element
- Handling collisions:
  - Chaining: store multiple items in a list at the same index
  - Open addressing: find another empty slot using a probing sequence
- Resize array when hash table becomes too full
- Details and complexity analysis may be intricate
- Amortized average-case complexity: $\Theta(1)$ for all operations

Hash Tables in Python

- set and dict implement hash tables internally
- Collisions are handled automatically

Python

```python
# Create dictionary
d = {}
d['apple'] = 5
d['banana'] = 3

# Access
print(d['apple'])  # 5

# Delete
del d['banana']
```

Hash Tables in C++

In the C++ STL, `unordered_map` and `unordered_set` are implemented as hash tables

C++

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

int main() {
  unordered_map<string, int> table;
  table["apple"] = 5;
  table["banana"] = 3;
  cout << table["apple"] << endl;  // 5
  table.erase("banana");
  return 0;
}
```

Comparison of Value-Based Data Structures

Data structure	Search	Insertion	Deletion	Ordered Traversal
Balanced binary search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	possible
Hash table	$\Theta(1)^*$	$\Theta(1)^*$	$\Theta(1)^*$	impossible

* amortized and usually average-case complexity; though possible to have $\Theta(1)$ non-amortized (linear hashing) and worst-case (perfect hashing)

Handling Duplicate Elements: Multisets and Multimaps

- Standard sets and maps do do not allow duplicates:
  - C++: `std::set`, `std::map`, `std::unordered_set`, `std::unordered_map`
  - Python: `set`, `dict`
- Sometimes we need to store multiple occurrences of the same element or key
- No need to change the underlying data storage techniques!
- Operations are similar to sets/maps, but duplicates are preserved
- C++:
  - `std::multiset`, `std::unordered_multiset`: allows multiple copies of an element
  - `std::multimap`, `std::unordered_multimap`: allows multiple values per key
- In Python, not directly supported but workarounds:
  - `collections.Counter`: counts occurrences of elements (like a multiset) – or use a `dict` whose values are the number of occurrences of elements
  - `dict` with `list` as values: emulate a multimap

Outline

Common Data Structures for Collections

Multi-File Programs

- Introduction
- In C/C++: Headers and implementation files
- In Python: Modules and Packages

Separating Complex Programs into Multiple Files

- Large programs quickly become difficult to manage if everything is written in a single file

Separating Complex Programs into Multiple Files

- Large programs quickly become difficult to manage if everything is written in a single file
- Separation into files allows:

Separating Complex Programs into Multiple Files

- Large programs quickly become difficult to manage if everything is written in a single file
- Separation into files allows:
  - Modularity: each file implements a well-defined component (e.g., a data structure, an algorithm)

Separating Complex Programs into Multiple Files

- Large programs quickly become difficult to manage if everything is written in a single file
- Separation into files allows:
  - Modularity: each file implements a well-defined component (e.g., a data structure, an algorithm)
  - Reusability: components can be reused in other projects without rewriting code

Separating Complex Programs into Multiple Files

- Large programs quickly become difficult to manage if everything is written in a single file
- Separation into files allows:
  - Modularity: each file implements a well-defined component (e.g., a data structure, an algorithm)
  - Reusability: components can be reused in other projects without rewriting code
  - Maintainability: changes or bug fixes can be made locally, without affecting unrelated parts

Separating Complex Programs into Multiple Files

- Large programs quickly become difficult to manage if everything is written in a single file
- Separation into files allows:
  - Modularity: each file implements a well-defined component (e.g., a data structure, an algorithm)
  - Reusability: components can be reused in other projects without rewriting code
  - Maintainability: changes or bug fixes can be made locally, without affecting unrelated parts
  - Collaboration: multiple developers can work simultaneously on different files

Separating Complex Programs into Multiple Files

- Large programs quickly become difficult to manage if everything is written in a single file
- Separation into files allows:
  - Modularity: each file implements a well-defined component (e.g., a data structure, an algorithm)
  - Reusability: components can be reused in other projects without rewriting code
  - Maintainability: changes or bug fixes can be made locally, without affecting unrelated parts
  - Collaboration: multiple developers can work simultaneously on different files
  - Compilation efficiency: only modified files need recompilation

Separating Complex Programs into Multiple Files

- Large programs quickly become difficult to manage if everything is written in a single file
- Separation into files allows:
  - Modularity: each file implements a well-defined component (e.g., a data structure, an algorithm)
  - Reusability: components can be reused in other projects without rewriting code
  - Maintainability: changes or bug fixes can be made locally, without affecting unrelated parts
  - Collaboration: multiple developers can work simultaneously on different files
  - Compilation efficiency: only modified files need recompilation
- Headers and implementation files in C/C++

Separating Complex Programs into Multiple Files

- Large programs quickly become difficult to manage if everything is written in a single file
- Separation into files allows:
  - Modularity: each file implements a well-defined component (e.g., a data structure, an algorithm)
  - Reusability: components can be reused in other projects without rewriting code
  - Maintainability: changes or bug fixes can be made locally, without affecting unrelated parts
  - Collaboration: multiple developers can work simultaneously on different files
  - Compilation efficiency: only modified files need recompilation
- Headers and implementation files in C/C++
- Modules and packages in Python

Outline

Common Data Structures for Collections

Multi-File Programs

Introduction

In C/C++: Headers and implementation files

In Python: Modules and Packages

Multiple files in a C or C++ program

- A typical program may split into separate files:

Multiple files in a C or C++ program

- A typical program may split into separate files:
    - Each definition of a class that is a major component of the implementation

Multiple files in a C or C++ program

- A typical program may split into separate files:
  - Each definition of a class that is a major component of the implementation
  - Each major function or collection of functions that serve related purposes

Multiple files in a C or C++ program

- A typical program may split into separate files:
  - Each definition of a class that is a major component of the implementation
  - Each major function or collection of functions that serve related purposes
  - The definition of the `main` function

Multiple files in a C or C++ program

- A typical program may split into separate files:
  - Each definition of a class that is a major component of the implementation
  - Each major function or collection of functions that serve related purposes
  - The definition of the `main` function
- However, we still have to respect a fundamental rule: every function, class, global variable, etc., needs to have been declared before being used

Multiple files in a C or C++ program

- A typical program may split into separate files:
  - Each definition of a class that is a major component of the implementation
  - Each major function or collection of functions that serve related purposes
  - The definition of the `main` function
- However, we still have to respect a fundamental rule: every function, class, global variable, etc., needs to have been declared before being used
- How to achieve this while separating code into multiple files?

Multiple files in a C or C++ program

- A typical program may split into separate files:
  - Each definition of a class that is a major component of the implementation
  - Each major function or collection of functions that serve related purposes
  - The definition of the `main` function
- However, we still have to respect a fundamental rule: every function, class, global variable, etc., needs to have been declared before being used
- How to achieve this while separating code into multiple files?
  - Declarations go into specific files (headers) that get included into implementation files which contain the implementations – the inclusion is done by a special tool that runs before the compiler, the preprocessor

Multiple files in a C or C++ program

- A typical program may split into separate files:
  - Each definition of a class that is a major component of the implementation
  - Each major function or collection of functions that serve related purposes
  - The definition of the `main` function
- However, we still have to respect a fundamental rule: every function, class, global variable, etc., needs to have been declared before being used
- How to achieve this while separating code into multiple files?
  - Declarations go into specific files (headers) that get included into implementation files which contain the implementations – the inclusion is done by a special tool that runs before the compiler, the preprocessor
  - Each implementation file is compiled separately into a compiled object file

Multiple files in a C or C++ program

- A typical program may split into separate files:
  - Each definition of a class that is a major component of the implementation
  - Each major function or collection of functions that serve related purposes
  - The definition of the `main` function
- However, we still have to respect a fundamental rule: every function, class, global variable, etc., needs to have been declared before being used
- How to achieve this while separating code into multiple files?
  - Declarations go into specific files (headers) that get included into implementation files which contain the implementations – the inclusion is done by a special tool that runs before the compiler, the preprocessor
  - Each implementation file is compiled separately into a compiled object file
  - All compiled object files are linked together to produce an executable (or a library, itself used by different executables)

Compilation process

Source files

.c or .cpp file #1 (implementation) → Preprocessor → Compiler → .o file (object code) → Linker

.h file (header) → both Preprocessors

.c or .cpp file #2 (implementation) → Preprocessor → Compiler → .o file (object code) → Linker

Linker → Executable program

Final output

What to Put in a Header File

- Header files (`.h`) are meant for declarations, not implementations
- They tell the compiler:
  - Which functions, classes, structures, constants, global variables exist
  - What are their types, their size in memory, the way they can be used (e.g., methods, properties)
- Typical contents of a header file:
  - Function declarations
  - Class and struct definitions
  - Global variable and constant declarations
- No function/method bodies or variable definitions (exception when the function definition is almost trivial, e.g., a constructor with simple behavior)
- In C++, headers also include template definitions (but we will not cover this); in that case, it is customary to name them .hpp

The C/C++ Preprocessor

- The preprocessor runs before the compiler
- It processes all lines starting with #:
  - `#include`: insert contents of another file
  - `#ifdef`, `#ifndef`, `#endif`: conditional inclusion
  - `#define`: define a symbolic macro – still commonly used, but avoid (prefer constant global variables or function definitions)
- The preprocessor performs a purely textual substitution step
- After preprocessing, the compiler only sees a single expanded source file

Preprocessor Inclusions

- The directive `#include` copies another file's contents into the current one

C++

```cpp
#include <iostream>  // System or standard library header
#include "myutils.h" // Local project header
```

- The angle brackets `<...>` tell the preprocessor to look in system include paths
- The quotes `"..."` tell it to look in the current directory first
- Standard C++ STL headers do not end with `.h`, standard C headers do
- Repeated inclusions of the same header may cause duplicate definition errors

Include Guards

- To avoid multiple inclusions of the same header, use include guards
- These are conditional preprocessor directives ensuring that the file is included only once
- Within a file `myutils.h`:

C

```c
#ifndef MYUTILS_H
#define MYUTILS_H

void print_message(const char* msg);

#endif // MYUTILS_H
```

Example: Multi-File Program (1/2)

File utils.h

C++

```
#ifndef UTILS_H
#define UTILS_H

void greet(const char* name);

#endif // UTILS_H
```

File utils.cpp

C++

```
#include "utils.h"
#include <iostream>

void greet(const char* name) {
  std::cout << "Hello, " << name << "!" << std::endl;
}
```

Example: Multi-File Program (2/2)

File main.cpp

C++

```
#include "utils.h"

int main() {
  greet("world");
  return 0;
}
```

Outline

Common Data Structures for Collections

Multi-File Programs
- Introduction
- In C/C++: Headers and implementation files
- In Python: Modules and Packages

Modules in Python

- A module is a single Python file (.py) that contains

Modules in Python

- A module is a single Python file (.py) that contains
  - Function definitions

Modules in Python

- A module is a single Python file (.py) that contains
  - Function definitions
  - Class definitions

Modules in Python

- A module is a single Python file (.py) that contains
  - Function definitions
  - Class definitions
  - Constants and global variables

Modules in Python

- A module is a single Python file (.py) that contains
    - Function definitions
    - Class definitions
    - Constants and global variables
- Modules allow:

Modules in Python

- A module is a single Python file (.py) that contains
  - Function definitions
  - Class definitions
  - Constants and global variables
- Modules allow:
  - Organization: group related code together

Modules in Python

- A module is a single Python file (.py) that contains
  - Function definitions
  - Class definitions
  - Constants and global variables
- Modules allow:
  - Organization: group related code together
  - Reusability: import and reuse in multiple programs

Modules in Python

- A module is a single Python file (.py) that contains
  - Function definitions
  - Class definitions
  - Constants and global variables
- Modules allow:
  - Organization: group related code together
  - Reusability: import and reuse in multiple programs
  - Namespace management: avoid polluting the global namespace

Modules in Python

- A module is a single Python file (`.py`) that contains
  - Function definitions
  - Class definitions
  - Constants and global variables
- Modules allow:
  - Organization: group related code together
  - Reusability: import and reuse in multiple programs
  - Namespace management: avoid polluting the global namespace
- Example: `utils.py` could define helper functions like `greet()`

Importing Modules in Python

- Use import to access functions, classes, and variables from another module

Python
```python
# main.py
import utils

utils.greet("world")
```

- You can also import only specific items:

Python
```python
from utils import greet
greet("world")
```

Packages in Python

- A package is a directory containing multiple Python modules
- It must contain an `__init__.py` file (may be empty)
- Packages allow:
  - Hierarchical organization of modules
  - Clearer namespace management
  - Easy distribution and reuse of larger code bases
- Example directory structure:

```
myproject/
  __init__.py
  utils.py
  math_helpers.py
```

Importing from Packages

- Modules inside a package can be imported using dot notation

Python

```python
from myproject import utils
utils.greet("world")

from myproject.math_helpers import factorial
print(factorial(5))
```

- You can also use aliases:

Python

```python
import myproject.utils as ut
ut.greet("world")
```

Main Function in Python

- Unlike C/C++, Python does not require a `main()` function
- However, it is a common convention to define one for clarity, especially when mixing with the use of modules

Python
```python
def main():
    greet("world")
```

- Use the special check to allow a file to be run as a script or imported as a module:

Python
```python
if __name__ == "__main__":
    main()
```

- This ensures that the script runs ‘main()‘ only when executed directly

Example: Python Multi-File Program (1/2)

File utils.py

Python

```python
def greet(name):
    print(f"Hello, {name}!")
```

File math_helpers.py

Python

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

Example: Python Multi-File Program (2/2)

File `main.py`

Python

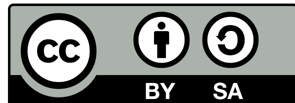
```
from utils import greet
from math_helpers import factorial

def main():
    greet("world")
    print(factorial(5))

if __name__ == "__main__":
    main()
```

Running python `main.py` prints:

```
Hello, world!
120
```

Licensing

This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.



Media used:

- Circular buffer diagram slide 11 is CC-BY-SA 3.0 by Cburnett
- Queue diagram slide 17 is CC-BY-SA 3.0 by Vegpuff
- Hash table diagram slide 35 is CC-BY-SA 3.0 by Jorge Stolfi