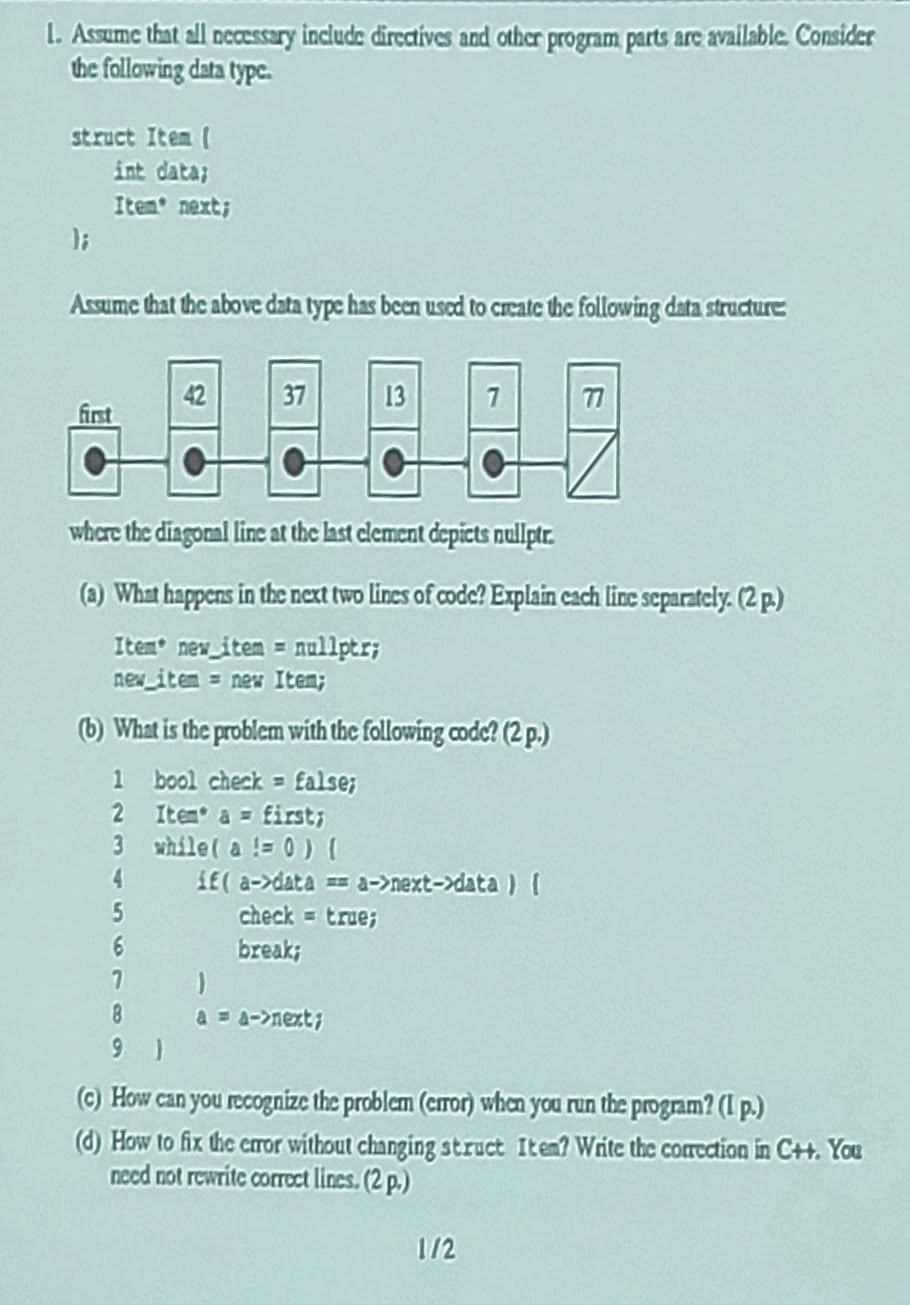
1. Assume that all necessary include directives and other program parts are available. Consider the following data type.

struct Item{

int data;  
 Item\* next;  
};

Assume that the above data type has been used to create the following data structure:

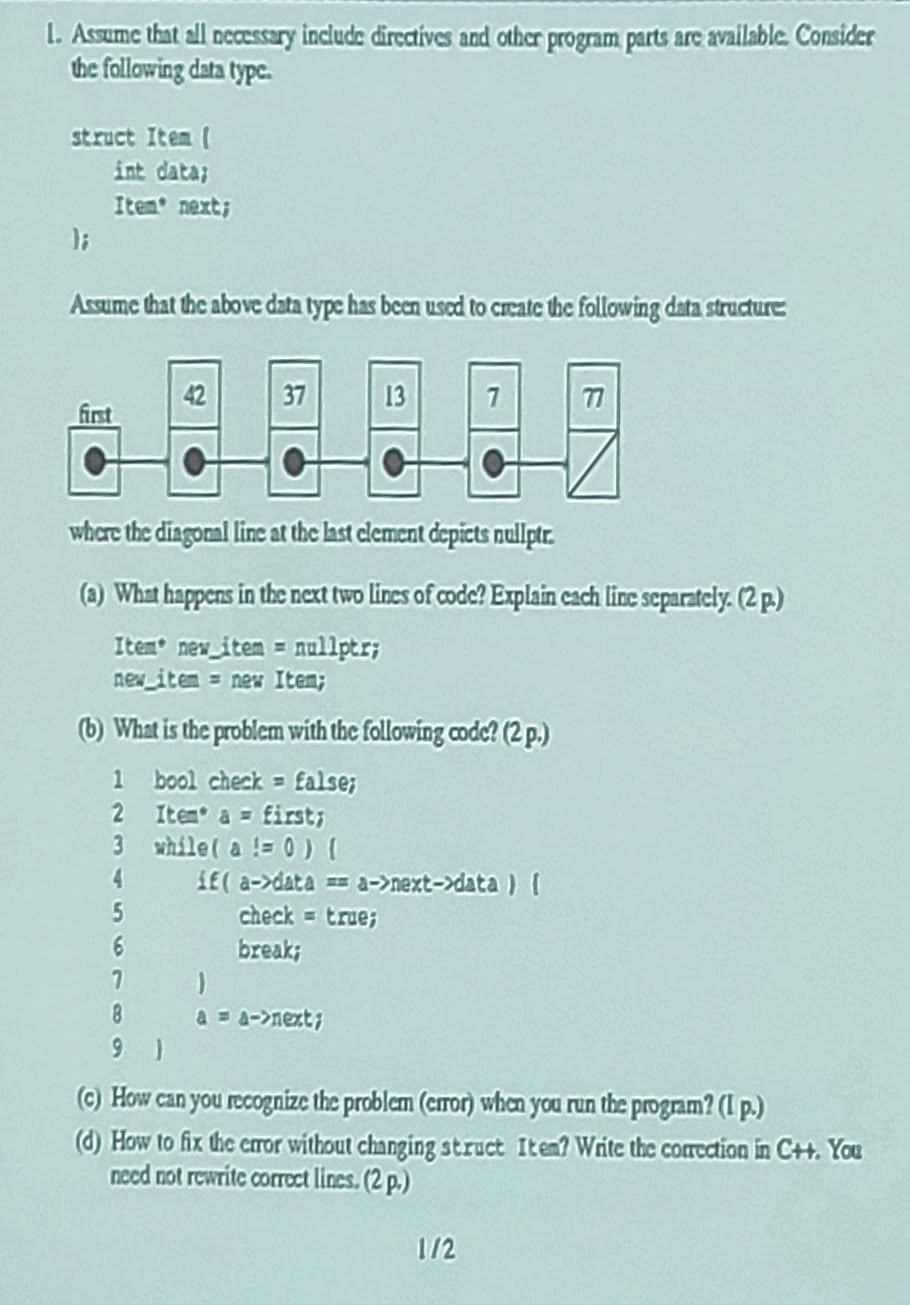


where the diagonal line at the last element depicts *nullptr*.

1. What happens in the next two lines of code? Explain each line separately

|  |
| --- |
| Item\* new\_item = nullptr;  new\_item = new Item; |

1. A pointer of Item data structure is created and assigned to null value
2. Such pointer now points into a certain address memory where the new structure data type is allocated
3. What is the problem with the following code?



1. How can you recognize the problem (error) when you run the program?

The program would crash or do nothing

1. How to fix the error without changing strict Item? Write the correction in C++. You need not rewrite correct lines

|  |
| --- |
| bool check = false;  Item\* a = first;  while(a != nullptr){  if(a->data == new\_data){  Item\* new\_item = new Item;  new\_item->data = new\_data;  new\_item->next = a->next;  a->next = new\_item;  break;  }  a = a->next;  } |
|  |

1. Write a piece of code that adds a new element to the above kind of data structure such that the new element is added after an element that has the same data value as the element to be added. If a suitable adding place is not found, nothing happens.

2.

(a) Write a recursive function that finds out how many times a certain character appears in a string. How does the call look like, when you call your function from the main program?

|  |
| --- |
| int fun(string str, char ch, int index){  if(str.length() <= index)  return 0;  if(str.at(index) == ch)  return 1+fun(str, ch, index-1);  else  return fun(str, ch, index–1); |
|  |

(b) Is your function tail-recursive? Why?

Not tail-recursive since the return value of the recursive call is not just composed by the returned function but also an additional operation.

3.

(a) What are the benefits of version control?

- Makes collaboration work with the same files possible without overwriting work that others have done

- Storing versions properly

- Restoring previous versions

- Keep track of the work and changes

- Backups

(b) What does the parent-child mechanism mean in graphical user interfaces?

It is related to the inheritance concept which is a mechanism to fowm new subclasses (or inherited classes) [child] from an existing one [parent]. The subclasses have the same properties as the base class but with some additional ones.

QObject <- QWidget <- QMainWindow

All the user interface elements in Qt have been inherited from the class QObject, whose public interface provides some very-low-level tools for handling user interface elements (widgets) (e.g. communication between different user interface elements with the signal-slot mechanism). QObject is the base class for the class QWidget. And QWidget is the base class of QMainWindow.

(c) What kind of data structure (constructed from STL and other C++ structures) would you use when implementing a program that lists the author index from the given text? The text (book) consists of several pages, and it contains references to (other) authors (and texts written by them). The program counts the amount of those pages that container reference to an auhor. Note that a single page may contain several references to the same author, but such occurrences will be counted only once. It must be possible to list the authors in alphabetical order. In addition, from each author you must list the page numbers, on which the author has been referenced.

map<string, set<int>>

4.

(a) Consider parameter passing in C++. Which different choices you have? Try to find as many of them as possible. Which situations each different way suits best?

* Value
* Reference (for large parameters)
* Const
* Default values (e.g. bool ignore\_empty = false as a parameter function)

(b) Give some concrete benefits that modularity provides you when implementing and managing large programs.

- The implementation of a module can be modified while the public interface stays intact

- The parts of the program that logically belong together can be combined in the same package, which simplifies the program and makes its testing and managing easier

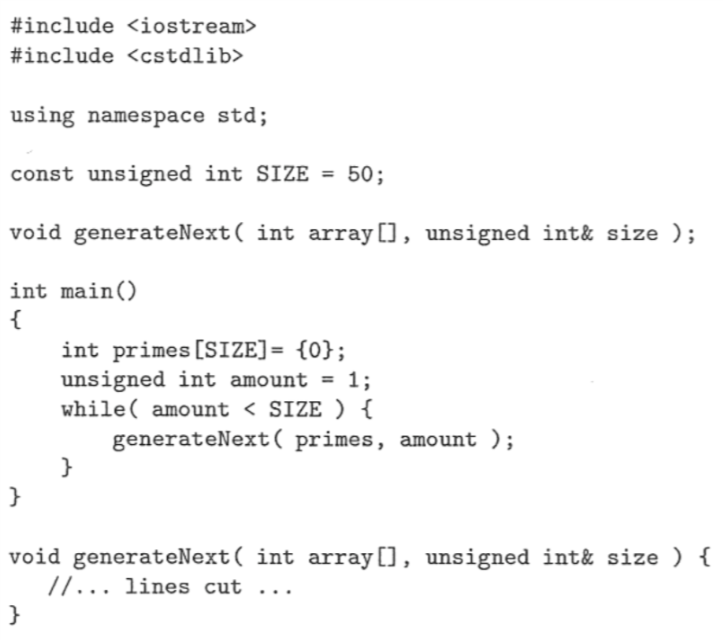
- The modules can be developed in the project side by side after agreeing on the public interface

- Modularity is a good tool in managing large programming projects

- Often you can reuse whole or partial modules

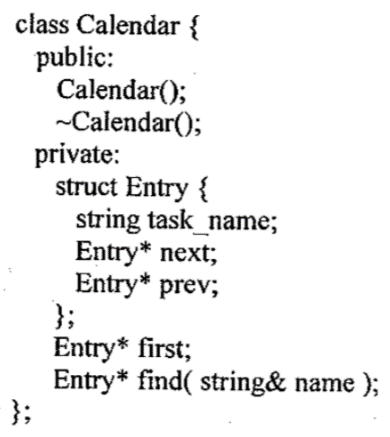
- Most programming languages that support modularity allow you to compile modules separately.This speeds up the developing, and uses less resources, since after completing the changes, you only need to re-compile the modules that were modified

1. What are functions used for in programming? Explain the key characteristics of functions in C++ by using the following code:



This piece of code is intended for filling an array of 50 different prime numbers.

1. The ADT Calendar has so far been defined as follows:

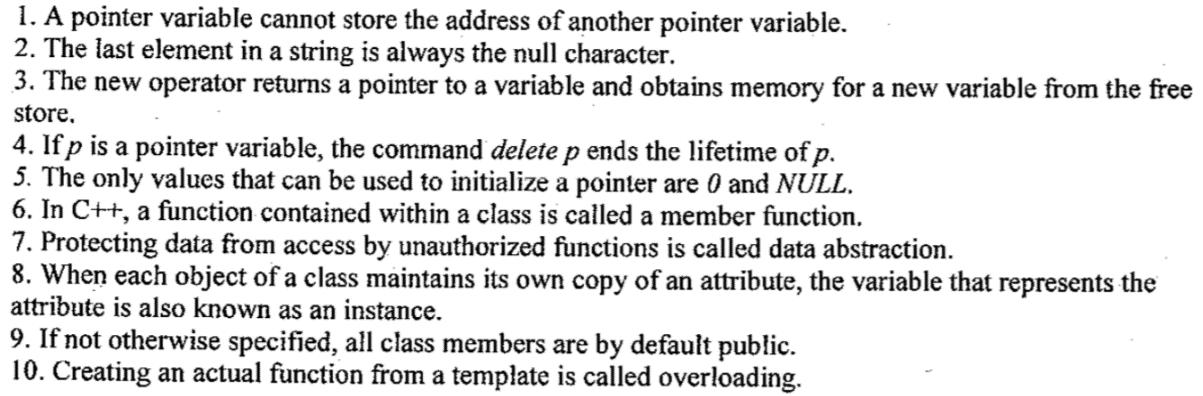


Add a specification and give an implementation of three member functions:

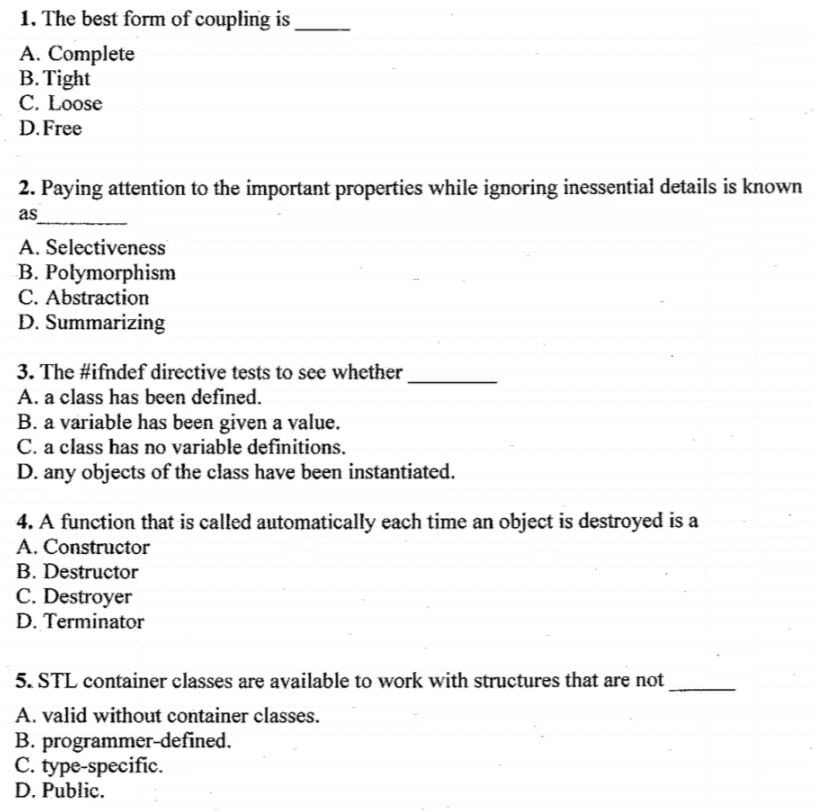
1. To add a new element to the list if it is not already found
2. To remove an element. The function should return false if the element is not found
3. To count the amount of elements with a specified name in the list

TRUE OR FALSE

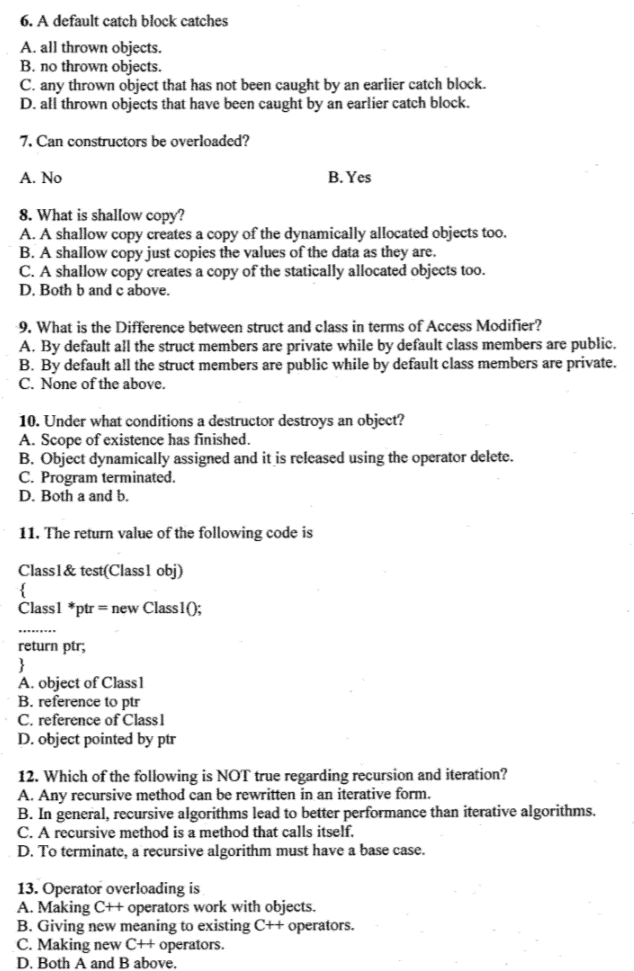
|  |  |
| --- | --- |
| 1 | T |
| 2 | T |
| 3 |  |
| 4 | T |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | T |
| 9 | T |
| 10 | T |



Function overloading is a feature in C++ where two or more functions can have the same name but different parameters



B. Destructor



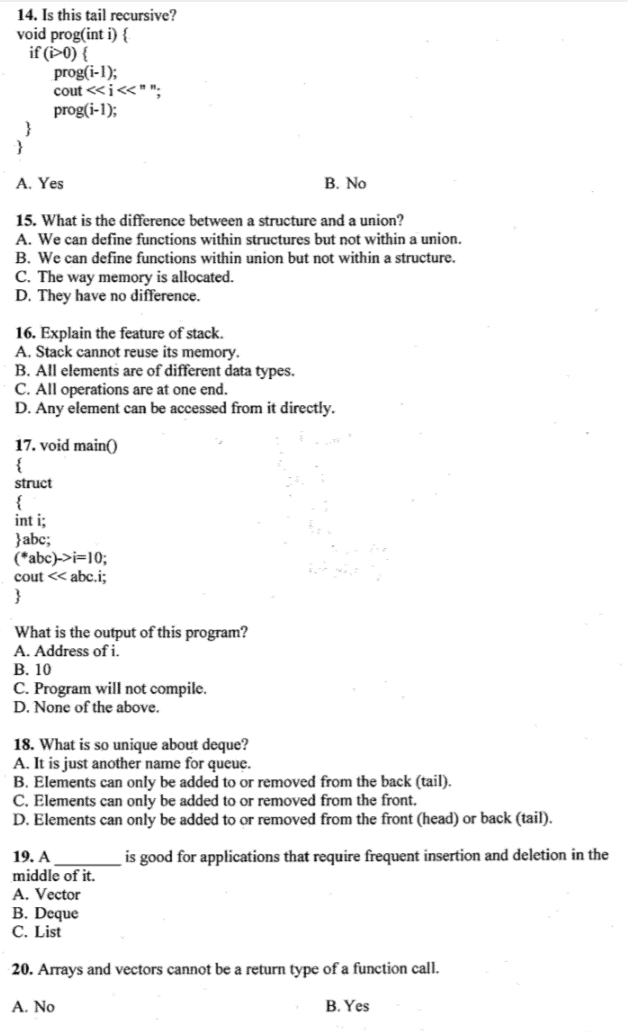


What is the difference between a Struct and a Class?

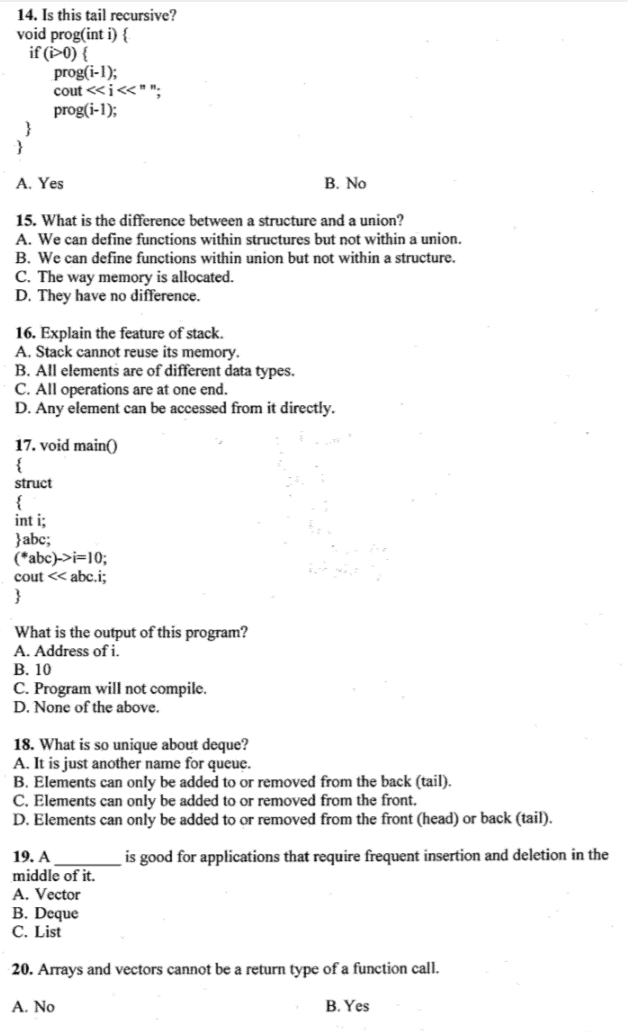
* a class can also contain functions (called methods)
* the member variables and methods are hidden from the outside world (private by default), unless their declaration follows a public label
* there can be a pair of special methods that are run automatically when an instance of the class (an object) is created or destroyed
  + the constructor
  + the destructor
* operators to work on the new data type can be defined using special methods (member functions)
* one class can be used as the basis for the definition of another (inheritance)
* declaring a variable of the new type (an instance of the class, an object) requires just the name of the class. the keyword class is not required
* struct: data container 🡪 small passive objects that carry public data and have no or few basic member functions
* class: complex data structure 🡪 bigger active objects that carry private data, interfaced through public member functions

When we create object-oriented code we want to separate the interface from the implementation 🡪 encapsulate the complicated implementation and only talk to the outside world through interfaces

Use setter / getter to access class’ private data









Explain shortly the following concepts

1. Reasons to use C++ Standard Template Library
2. Code reuse, no need to re-invent the wheel
3. Efficiency (fast and use less resources). Modern C++ compiler are tuned to optimize for C++ standard library code
4. Accurate, less buggy
5. Terse, readable code; reduced control flow
6. Standardization, guaranteed availability
7. A role model of writing library
8. Good knowledge of data structures and algorithms
9. Benefits of using const
10. Guards against inadvertent write to the variable
11. Self documenting
12. Enables compiler to do more optimization, making code tighter
13. const means the variable can be put in ROM

Until now, we have used only automatic variables, which means that allocating (reserving) memory for them and afterwards deallocating (releasing) it was automated:

* + At the variable definition, the compiler has taken care of finding the necessary amount of memory somewhere and allocate it.
  + When a variable reached the end of its lifetime, the compiler, again, has deallocated the memory that was no longer needed.

When the control over the lifetime of a variable is completely the programmer’s responsibility, the variable is called a dynamic variable. The mechanisms and tools used for controlling dynamic variables are called dynamic memory management.

new and delete commands are for allocate and deallocate dynamic memory

new:

* + The lifetime of a variable begins at the moment when new succeeds to allocate memory for the variable
  + A dynamic variable does not have a name, but the operation new returns a pointer, the value of which reveals where the new dynamic variable is located within the main memory
  + To make the dynamic variable useful, its address (the return valuw of new) needs to be stored in a pointer variable.
  + If the variable is a class-type one, new will take care of calling the constructor

delete:

* + The variable reaches the end of its lifetime and the memory that was allocated to it is deallocated for other uses
  + If a class-type variable has a destructor defined, delete will call the destructor

If you do not deallocate memory, the program will have memory leaks. It means that the program continues to keep memory locations allocated even though it no longer needs them. It is a bad thing especially for programs with long execution times since the amount of main memory used by the program increases burdening the whole system.

1. Shallow copy

A mechanism where you only copy the memory address (which often leads to trouble)

1. Valgring

It is a program tool executed separately which traces memory management errors. It analyzes the code of the program and prints error messages if:

* + your program allocates dynamic memory with new but does not deallocate it with delete
  + your program uses a variable (allocated dynamically or automatically) without a value set to it
  + you try to use dynamically allocated memory that has been deallocated already

1. Smart pointers

They are library data types that automate the deallocationg of memory when nothing points to it anymore

* + #include <memory>
  + shared\_ptr
  + unique\_ptr
  + weak\_ptr

Note: Dangling pointer is a pointer to an object which has been already deleted

1. Function pointers

You can pass a function as parameter to another function, pass it as a return value of a function or store it into a variable. A function taking another function as its parameters or returning another function is called a higher-order function.

* + We want to choose the functionality to be executed in a program but we do not want to include a very long if structure with each block containing one of the possible functionalities
  + We are implementing graphical user interfaces and we want to bring a functionality to a component of the user service

1. Abstract data type

It means a data type that is defined by its possible values and operations handling these values. So that, the type is considered by the provided operations.

1. Node of a linked structure

Linked lists consist of struct-type elements, each separately allocated (with the command new). In addition to the actual value stored in the node of a linked structure, each element includes a pointer pointing to the next list element. (Sometimes it is also efficient to have a second pointer variable to store the address of the last element in the list)

1. Modularity

It is a mechanism that divides a large program into small, more easily manageable parts when designing and implementing a program. If a program is modular, it has been divided into distinctly considered parts, and the result of their combined functioning is the final program.

Module is a whole consisting of cohesive programming structures. Each module is implemented as a separate source code file. Each module has a public and private interface but they are different than classes.

1. Recursion

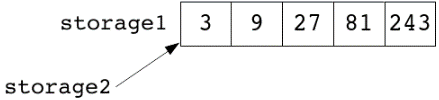
* A function is recursive if it calls itself (directly or indirectly)
* During the execution, as many instances of the recursive function are on as there are unfinished recursive calls
* Each instance has its own values for the parameters and local variables
* They have 2 essential characteristics:
  + It has a terminatic condition (or multiple) that recognizes trivial cases and reacts to them without having to make a new recursive call
  + Each recursive call has to simplify the problem in order to finally reach the trivial case
* It is shorter and cleaner than a loop structure BUT slightly slower and uses more memory
* Types
  + **Direct recursion**: when calls itself in its own body
  + **Indirect or mutual recursion**: a function calls another
  + **Tail recursion**: the return value of a recursive call becomes the return value of the calling instance (function) without any additional operations. After the call, there are NO statements to execute or expressions to evaluate.
    - You can mechanically move it to a loop structure

1. Pointers vs References

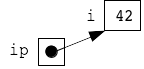
Pointers are variables with can store memory addresses of a variable. They have to be declared first.  
  
References are another names for an existing variable, once this variable is initialized, the variable name can be used to refer to another variable.

|  |  |  |
| --- | --- | --- |
|  | C++ Reference | Pointer |
| Variables | It is an alias to an existing variable. The main use of this variable is working as a parameter which works as a pass-by-reference. This is passed to a function. The function to which this is passed works on the original variable instead of the copy in pass by value. The changes made inside the function will also be reflected outside. | Are variables that store address, it is a another data type like int, double, char… |
| Declaration | &  The reference variable’s values can be changed in original and copy of variable | \*  It should be declared in advance |
| Reassignment | Cannot | Yes, it can. Handy property when implementing data structures like linked lists, trees,… |
| Memory address | It shares the same address as the original variable. It never points to a new variable until the old is deleted or goes out of the scope | Has its own memory address. It can have new values assigned to itself |
| Null value | Cannot have it | Can have it |
| When to use | Indirectly accessing a variable | Used to implement data structures |

Example of a reference:



Example of a pointer:



1. Interpreted vs compiled programs

Interpreted programs 🡺 require of an utility program every single time we want to run the program (interpreter). It transforms the commands in the source code to the machine language as the program’s execution progresses

Compiled programs 🡺 require of a compiler which processes the source code completely before it is ever executed. It produces a whole machine language program that is stored in a separate file (.exe in windows). Once the compilation is done, the compiler is not needed anymore. But if the source code is ever modified, then it should be compiled again so that we can run the updated version. The compilation phases are:

* + Compilation into an object file, which is in machine code but it cannot be executed yet since it needs parts from the library and thus the program must be linked
  + Linking means that separately compiled parts are joined together as a single executable program. Usually this phase is hidden but large programs require compilation made in two differentiated phases

1. Static vs Dynamic typing

In dynamic typing (Python, interpreted programming languages) the vadility of the data is only checked when we try to use the data. In practice, dynamic typing in a programming language results:

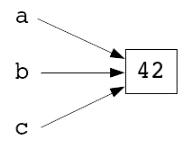
* One variable can contain different type values during the execution program
* The program will run fine until we reach the point where we try to do something with values that are not compatible

In static typing (C++), the vadility of the data is checked during compilation. In practice, results in:

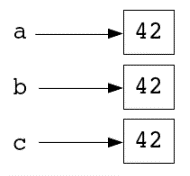
* Variables must be defined before they can be used
* Variables have a permanent type and only the data that is of the same type can be stored in a the variable
* If there is an error in the program related to types, the compiler produces an error message and the program will not be compiled. Therefore it cannot be executed.

1. Value vs Reference semantics

In Reference semantics, there is only one data element which can have one or more names



In Value semantics, the processed data element is copied during initialization and assignment



1. STL (Standard Template Library)

It is a combination of:

* + Containers
    - Sequence containers (a container that preserves the order of the elements)

The difference by using [] or .at operators is that if an element is out of bounds:

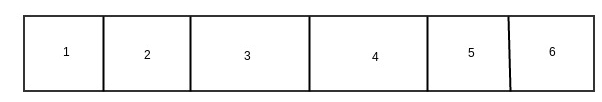
* + - [] operator aborts with an error message
    - at() throws an exception

They do not have find() member function since it is too slow.

* + - * Vector

Stores elements at contiguous memory locations like an array. Random access is possible (that is, we can directly access the an element using operator []). The reallocation is expensive, it grows exponentially (vector.push\_back()).

1. Fast insert/remove at the end
2. Slow insert/remove at the beginning or in the middle
3. Slow search



* + - * Deque

Elements are NOT contiguous in memory. The growth is linear with a fixed size. No reallocation issues but slightly slower than vector

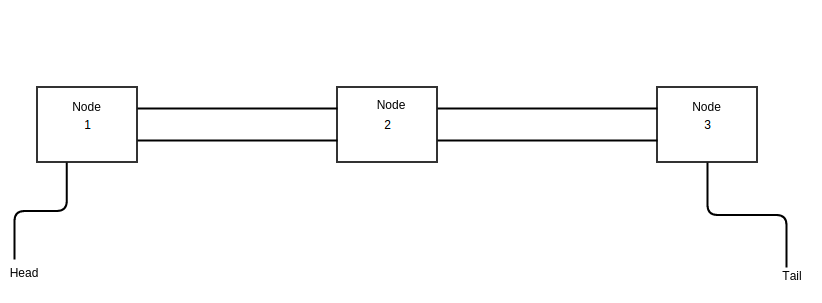
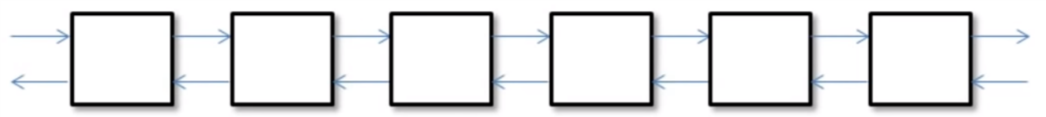


1. Fast insert/remove at the beginning and the end
2. Slow insert/remove in the middle
3. Slow search
   * + - Lists

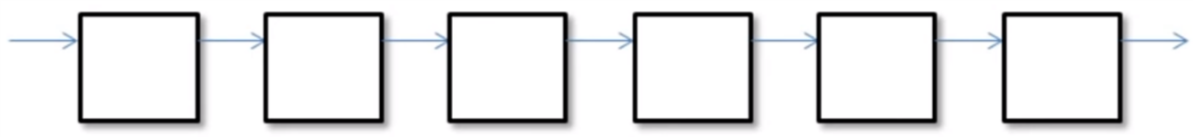
Stores elements at non contiguous memory location (internally uses a doubly linked list). In order to access a certain element, we need to iterate through the first previous elements one by one. Random access is NOT possible

Useful and more efficient than vectors when inserting and deleting at start or middle.

Doubly link list: forward and backward pointers



1. Fast insert and remove at any place in the list faster than vector/deque
2. Slower search than vector/deque
3. No random access, no [] operator  
   * + - Forward list: forward pointers



* + - * Arrays

1. Size cannot be changed
2. Two arrays of the same data structure can be of different type if the size differs
3. They canNOT use begin(), end(), size() and swap()
   * + Associative containers (after initialization, they are empty)

Stored elements in a way that is fast as possible to search. They are not arranged in a queue and therefore do not have index. The data is sored in associative containers 🡪 (search) key (searches are conducted with that key).

They are sorted, default criteria is <

* + - * Set: each value can exist no more than once  
        set<int> prime\_numbers = {2, 3, 5, 7, 11, 13, 13, 17};  
        my\_set.insert(3);

my\_set.insert(1); // {1, 3}

my\_set.insert(7); // {1, 3, 7}

ship\_has\_been\_loaded.insert(word)

friends.erase(“Teppo”)  
ship\_has\_been\_loaded.clear()

not\_friends.empty()

* + - * Map (analogue to dict): pairs of a key and a mapped value  
        It does not allow duplicated keys  
        map<char, int> mymap;  
        mymap.insert(pair<char,int>(‘a’,100));  
        mymap.insert(makepair(‘z’, 200));  
        map<char,int>::iterator it = mymap.begin()  
        mymap.insert(it, pair<char,int>(‘b’,300));

map<string, double> prices = {{“milk”, 1.05}, {“cheese”, 4.95}};

if ( dictionary\_3.find(word) != dictionary\_3.end() ) {

cout << dictionary\_3.at(word) << endl;

} else {

cout << "Error, an unknown keyword!" << endl;

}

cout << prices.at("milk") << endl; // 1.05

if ( prices\_2.erase("chocolate") ) {

// The erasing was successful, "chocolate"

// is not in the pricelist anymore.

...

} else {

// The erasing was not successful, "chocolate"

// was not in the pricelist to start with.

...

}

cout << "Dictionary contains " << dictionary\_2.size()

<< " pairs of words." << endl;

It is equivalent to a struct:

struct {

string first; // key

double second; // mapped value

};

int main() {

map<int, string> students = {

{ 200001, "Teekkari, Tiina" },

{ 123456, "Teekkari, Teemu" },

// ···

};

map<int, string>::iterator iter;

iter = students.begin();

while ( iter != students.end() ) {

cout << iter->first << " "

<< iter->second << endl;

cout << (\*iter).first << " "

<< (\*iter).second << endl;

++iter;

}

// or alternatively

for ( auto data\_pair : students ) {

cout << data\_pair.first << " "

<< data\_pair.second << endl;

}

}

This would print the elements in ascending order. The elements are ordered according to the key values

* + - * Pairs:

#include <utility>

pair <int, char> pair1;

pair1.first = 1;

pair1.second = 'a';

pair <int, char> pair2 (1, 'a');

pair <int, char> pair3;

pair3 = make\_pair(1, 'a');

auto pair4 = make\_pair(1, 'a');

* + - Unordered containers 🡪 unordered\_map, unordered\_set

In the cases that there is no need to keep the elements in order, it is possible to make searching, inserting and removing elements more efficiently

* + Iterators

Data types that you can examine and modify the elements stored in a container. This solves the problem on how to go through all the elements within a container.

In case of a vector, it is not a problem because the elements of a vector can be indexed through a loop. But in other cases that elements do not have an index number, iterators are required.

One iterator allows you to mark the location of one element. Two iterators, to show a range of elements.

* + - Random access iterator: vector, deque, array

vector<int> itr;

itr = itr + 5;

itr = itr - 4;

if(itr2 > itr1)

++itr;

--itr;

* + - Bidirectional iterator: list, set, map

list<int> itr;

++itr;

--itr;

* + - Forward iterator: forward\_list

forward\_list<int> itr;

++itr;

* + - Input iterator: read and process values while iterating forward

int x = \*itr;

* + - Output iterator: output values while iterating forward

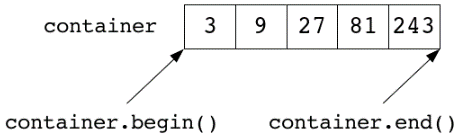
\*itr = 100

set<int>::iterator itr; 🡪 whatever is in the right side of ::, is a nested class  
 set<int>::const\_iterator citr; 🡪 read\_only access to container elements

advance(itr, 5) 🡪 itr+=5;  
 distance(itr1, itr2)

* + Algorithms

It is a set of the most common operations you need to conduct on elements within a container. They are able to operate with any STL container. They are mostly loops.



* + - count(vector.begin(), vector.end(), “counted value”) [it does not work in a that straightforward way for map structures
    - min\_element and max\_element

vector<int> amounts;

...

vector<int>::iterator smallest\_it;

smallest\_it = min\_element(amounts.begin(), amounts.end());

cout << "Smallest amount: " << \*smallest\_it << endl;

* + - find: searches the container for the chosen value and returns the iterator into the first element it finds or, if it cannot find the value, it returns the end iterator of a container.

vector<string>::iterator iter;

iter = find(patients.begin(), patients.end(), "Kai");

* + - replace: replaces all chosen values with a new value. It does NOT work with set or map structures

replace(text\_vector.begin(), text\_vector.end(), "TUT", "TUNI");

* + - fill: changes all elements within the iterator range into the given value

string my\_string = "";

fill(my\_string.begin(), my\_string.end(), '?');

* + - reverse(vector.begin(), vector.end()): reverses the order of the given range. It does not work with the set or map structures at all
    - shuffle: shuffles the elements and puts them in a random order. It does not work with the set or map structures at all

vector<Card> deck;

minstd\_rand gen; // A random number generator gen is created

shuffle(deck.begin(), deck.end(), gen);

#include <random>

* + - copy: copies the elements of the container into another one

string word = "";

vector<char> char\_vector(word.length());

copy(word.begin(), word.end(), char\_vector.begin());

* + - sort(students.begin(), students.end(), compare\_ids): sort data into order by magnitude

vector<int> vec = {9, 1, 10, 2, 45, 3, 90, 4, 9, 5, 8};  
sort(vec.begin(), vec.end()); // vec: 1 2 3 4 5 8 9 9 10 45 90

bool lsb\_less(int x, int y){  
 return (x%10)<(y%10)  
}  
sort(vec.begin(), vec.end(), lsb\_less); //sort with lsb\_less()  
// vec: 10 90 1 2 3 4 45 5 8 9 9

vector<int> vec = {4, 2, 5, 1, 3, 9};  
vector<int>::iterator itr = min\_element(vec.begin(), vec.end()); // itr->1  
  
sort(vec.begin(), itr); // vec: {2, 4, 5, 1, 3, 9} 🡪 algorithm always process ranges in a half-open way\_ [begin, end)  
  
reverse(itr, vec.end()); // vec: {2, 4, 5, 9, 3, 1} itr->9

vector<int> vec2(3);

copy(itr, vec.end(), // source (from itr to vec.end())  
 vec2.begin()); // pasted at vec2.begin()

vec3.insert(vec3.end(), itr, vec.end()); // insert in vec3.end() a vec from itr to vec.end()

par 🡪 odd

impar 🡪 even

bool isOdd(int i){

return i%2;

}  
  
int main(){

vector<int> vec = {2, 4, 5, 9, 2};

vector<int>::iterator itr = find\_if(vec.begin(), vec.end(), isOdd); // itr->5

In the case of arrays, iterators are behaved as pointers

int arr[4] = {6, 3, 7, 4};  
 sort(arr, arr+4);

* String

s1 = “Goodbye”;  
s1[2]; // ‘o’  
s1[2] = ‘x’; // Goxbye  
s1.at(2) = ‘y’ // Goybye  
s1.at(20) ; // throw exception of out\_of\_range  
s1[20]; //undefined behavior  
  
s1.front(); // ‘G’  
s1.back(); // ‘e’  
s1.push\_back(‘z’);  
s1.pop\_back();  
s1.begin();  
s1.end();  
//Like vector, string does not have push\_front() and pop\_front()  
  
string s3(s1-begin(), s1.begin()+3); // s3: Goo  
  
//Ranged access  
//assign, append, insert, replace  
string s2 = “Dragon Land”;  
  
s1.assign(s2); // s1 = s2;  
s1.assign(s2, 2, 4); // s1: agon  
s1.assign(“Wisdom”); // s1: Wisdom  
s1.assign(“Wisdom”, 3); // s1: Wis  
//s1.assign(s2, 3); // error  
s1.assign(3, ‘x’) // s1: “xxx”  
s1.assign({‘a’, ‘b’, ‘c’}); // s1: abc  
  
s1.append(“ def”); //s1: abc def  
s1.insert(2, “mountain”, 4) // s1: abmounc def  
s1.replace(2, 5, s2, 3, 3); // s1: abgon def  
  
s1.erase(1, 4); // s1: a def  
s1.replace(2, 5, s2, 3, 3); // s1: abgon def  
  
s1.erase(1, 4); // s1: a def  
s2.substr(2, 4); // agon  
s1 = “abc”;  
s1.wap(s2); // swap string contents

getline(cin, s1); // default delimiter of ‘\n’  
getline(cin, s1, ‘;’); // delimiter is ‘;’

# Orienteering feedback

==========================================================================

PROGRAM STRUCTURE: 10/10 points

+ The program is split into appropriately sized functions.

==========================================================================

CLASS INTERFACE AND USED DATASTRUCTURE: 15/15 points

+ The selected containers fit their tasks.

\* The decision to split the implementation of the routes module into

multiple modules is weird and feels forced. The Point, Points and Route

classes are so simple that they could be structs just as well, with the

code implementing the various features placed inside the Routes class.

This is especially true of the Point class - the class basically consists

of a bunch of getter functions and not much else. This only serves to

make the program structure more confusing while bringing basically no

value in return. It's good to practice object-oriented programming,

and your enthusiasm is to be admired, but don't overdo it. :)

\* The decision to store the map size in the Points class is confusing -

you should probably rename the class to something along the lines of

Map or whatever.

\* I would recommend that you read about memory ownership online. You aren't

expected to know this yet, but generally speaking, when an object reserves

memory the way your Points class does, that object becomes the owner of

the memory. Memory should, as a rule, always be released by the owner -

if other objects need to be able to control this behavior, the class

owning the memory should implement a function that enables other code

to request the object to release memory.

Something to consider is a situation where an object frees

another object's memory from outside - the object owning the memory has

no idea this has happened and is in an illegal state and if any reference

to released memory is made, the program will crash and this kind of

crash can often be difficult to debug.

So, to clarify, reserving memory in Points::add\_point and releasing it

in Routes::~Routes is not good - you'll want the owner of the memory, i.e.

the Points object, to release memory owned by itself. What I would do is

implement a destructor for the Points class that releases memory,

then calling that destructor from the destructor of the Routes class.

==========================================================================

GENERAL SOLUTION: 15/15 points

+ Large objects are passed by reference.

+ No memory issues other than the ones discussed above.

+ Variables are initialized.

\* Program code becomes clearer and easier to read, when tasks are ordered

appropriately and written compactly. For example, the outer else block in

Routes::greatest\_rise is completely redundant because the function contains

no code placed outside said else block.

\* Contrary to what you claim in a comment in routes.hh, the code template

does NOT force one to use raw pointers - they are mentioned nowhere in

the Routes class interface or definition, and even if they were, you could

still use smart pointers in your implementation and simply return raw

pointers for the user of the module whenever they call a function that

returns a pointer.

==========================================================================

PROGRAMMING STYLE: -13/-20 points

+ Variable and function names are mostly self-explanatory and consistent.

- An exception to the above would be the member grid\_ of the Points class -

it's a one-dimensional vector and as such it most definitely doesn't look

like a grid to the reader. This part would definitely require at least a

comment to explain what's going on, as do all non-self-explanatory

container variables.

+ The code positioning is neat.

\* There were some shortcomings in the comments:

- At the beginning of every source code file, there should be

a comment (so called file comment), which describes the meaning

and usage of the module, and information about the file author.

- At the beginning of each function/method, there should be a comment

describing its usage, parameters and return values.

\* This so-called "interface documentation" is usually written only

in the header file (.hh or .h) of each module, right before each

function/method declaration. Please notice, that there's no need to

copy the same comment in the source files (.cc, .c). Similarly, no

interface documentation is written for the main function, since its

role and parameters are usually the same anyway.

- Comments should describe the purpose of all such parts of the program,

that are not self-evident.

- Your custom classes have practically no comments at all. You should at

the VERY LEAST add a comment to each module explaining what the module

does so the reader doesn't have to keep guessing.

- Some of the comments either don't make sense or are outdated.

For example this bit in routes.hh:

//see photo

// put map<int [it is the rise], set<string>>

// the last element is the highest rise since it is orderded

// or .rbegin()

\* Remember that comments are extremely important - even the best code ever

is totally useless if no one knows how to use it.

\* Aim to keep code and comment lines under 80 characters long.

This improves the readability of the code.

\* There were some problems in the usage of the module mechanism:

\* According to appropriate programming principles, while "including"

self-made modules along with modules from the standard libraries,

self-made modules should come first.

\* You are importing unnecessary libraries in the header files - you

should only include libraries required to define the header file.

Libraries only needed for the implementation of said module should

be included in the .cpp, i.e. implementation, file.

\* "Using namespace std;" is bad practice and should be avoided. You can

do for example "using std::vector;" to refer to std::vector as just

"vector".

==========================================================================

VERSION CONTROL: 10/10 points

+ Good number of commits.

+ Fairly good commit messages, although something like "Routes printing

implementation done" would be better formulated in a more detailed

manner, for example "Implement Routes::print\_route".

==========================================================================