1**) what is structural padding and packing?**

'

It is a way to speed up CPU optimization.

Example:

#include <iostream>

using namespace std;

struct Base

{

char a;

char b;

int c;

};

int main()

{

cout<<sizeof(Base)<<endl;

cout<<sizeof(Cbase)<<endl;

return 0;

}

|a|b|-|-|c|c|c|c|

0 1 2 3 4 5 6 7

How data gets memory slots:

1Byte -> can be stored at multiple of 1 memory slot

2Byte -> can be stored at multiple 2 memory slots

4Byte -> can be stored at multiple of 4 memory slots

8Byte -> can be stored at multiple 8-memory slots

so, assume our memory locations are 0 to 10.

char can store multiple of 1bytesmeans it can store 0 to 10 in any position.

int is 4 bytes means it can store 0, 4, and 8 positions only.

double is 8 bytes means it can store multiple 8 means it starts 0, 8 position only.

so the above Example

|a|b|-|-|c|c|c|c|

0 1 2 3 4 5 6 7

Example: The structure is changed like below

struct Base

{

double a;

char b;

int c;

};

now the size is 16 because

|a|a|a|a|a|a|a|a|b|-| - | - | c | c | c | c |

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

In the above example, double is 8 bytes, so it starts from 0 positions after char is 1 byte, so it stores at 9 positions and int is 4 bytes, so it starts position are multiple of 4 means 0,4,8 but this is already filled and next position is 12 so compiler stores at 12 position. due to that the size is 16 if we use double c instead int c in the above example then the size is 24 because double stores from multiple of 8 bytes which means it starts this are already freed next are 16 so the compiler stores at 16 positions.

2) How compilation works internally in C And C++ ?

[Source Code] 1.cpp ---> Compiler ---> 1.o [object code (\*.o file)]-- |

[Source Code] 2.cpp ---> Compiler ---> 2.o [object code (\*.o file)]--\* --->[Linker] --> [Executable] --->

|

[Source Code] 3.cpp ---> Compiler ---> 3.0 [object code (\*.o file)]--\*

|

Library file ----

------------------------------------- Diagram 2 -------------------------------------------------------

Editor or IDE (write source code (.cpp, .h))----> Prepocessor (\*.i, include file, Replaced symbols) --->compiler (\*.s assmble code) ---> Assembler(\*.o object

code) ---> Linker --------------------------> .exe ---------> Loader ----> running Executable in memory

/|\ /|\

| |

| |

| |

static library(\*.lib, \*.a ) dynamic library

1) the first step is a preprocessor, here it replaces all the included library into code and removes the comments from the source code and replace the macro and inline function with the code in the preprossor state. then it generates \*.i files.

2) compiler stage, here it check the errors(tokenizer, anlytical error, symentic error) and generates the assemble code \*.s files.

3) assembler generates the \*.o files or object files.

3) How do delete[] know how many objects to delete?

1) over allocation 2) associative array

#include <iostream>

int const n = 10;

class Base

{

public:

int b\_var;

};

int main()

{

Base \*bp = new Base[n];

//char \*tmp = (char\*) operator new[](WORDSIZE+n\*sizeof(Base));

//Base\* p = (Base\*) (tmp+WORDSIZE);

//\*(size\_t\*)tmp = n;

//for(int i = 0 ; i < n ; i++)

// new(p+i) Base();

delete[] bp;

//size\_t n = \*(size\_t\*)((char\*)bp - WORDSIZE);

// while(n-- != 0)

// (bp+n)->~Base();

// operator delete[] ((char\*)bp - WORDSIZE);

}

4) What is the difference between a Reference and pointer?

1) Memory Address

2) Reassignment not possible

3) NULL value

4) Arithmetic operations

5) Indirection

5) what is a function pointer?

The normal pointer variable stores the address of another variable. function pointer stores the address of another function.

Example :

#include <iostream>

using namespace std;

int add(int val, int val2)

{

return (val+val2);

}

int fun1(int (\*fun)(int,int))

{

int sum = fun(1,2);

cout<<sum<<endl;

return sum;

}

int main()

{

int sum = fun1(add);

cout<< sum<<endl;

return 0;

}

6) **what is the use of the friend function and class in c++?**

The friend keyword is used to give access explicitly.

use case: If the Admin class one function needs Employee class information, then we can declare that function as a friend inside the employee class.

If one class is derived from some other class but another class fun needs this class information, then instead of deriving that class you can use the friend keyword.

What are the limitations of the friend function?

The major disadvantage of friend functions is that they occupy the maximum size of the memory and can’t do any run-time polymorphism concepts.

Example :

#include <iostream>

using namespace std;

class sample

{

int x;

public:

sample(int x):x(x){}

~sample() { }

friend void fun(sample obj);

};

void fun(sample obj)

{

cout<<"before fun : "<< obj.x << endl;

obj.x = 20;

cout<<"after fun : "<<obj.x <<endl;

}

int main()

{

sample obj(10);

fun(obj);

return 0;

}

7) How to stop someone from taking the address of your object?

overload the & operator and keep it private.

delete & operator from your class.

Example:

#include <iostream>

using namespace std;

class sample

{

private:

sample\* operator&()

{

cout<<"Bingo"<<endl;

return this;

}

public:

sample\* operator&() = delete;

};

int main()

{

sample obj;

sample \*obj1 = &obj;

cout<<&obj<<" "<<obj1<<endl;

return 0;

}

8) what is a void and void pointer in c++?

Topic: void

i) void is used to denote nothing

ii) If some function is not returning anything, we use void type to denote that.

iii) If some function doesn't take any parameters, then we use void type to denote that.

iv) we cannot create a void variable.

v) size of the void is 1 in gcc compiler but it is not valid to check size of void

TOPIC: void\*

i) void\* is a universal pointer.

ii) we can convert any data type pointer to void\* (except function pointer, const, or volatile)

iii) void\* cannot be dereferenced.

Bottom line :

void means nothing and void\* means a pointer to anything (except function pointer, const or volatile)

USE CASES:

//malloc and calloc returns void\* so that we can typecase to our desired data type.

//void\* are used to create a generic function in c.(compare function used in qsort function in c).

**9) static\_cast in c++ ?**

1) It performs implicit conversion between types.

Example:

#include <iostream>

using namespace std;

int main()

{

double d =20.0;

int a = d; // implicit conversion

int b = static\_cast<int>(d); // static\_cast

cout<<a<<" "<<b<<endl;

return 0;

}

2) use static\_cast when conversion between types is provided through conversion or conversion constructor.

Example:

#include <iostream>

using namespace std;

class Int

{

int x;

public:

Int(int x):x{x}

{

cout<<"conversion constructor : "<<endl;

}

operator string()

{

cout<<"conversion operator : "<<endl;

return to\_string(x);

}

};

int main()

{

Int obj(10);

obj = 20;

string str1 = obj;

string str = static\_cast<string>(obj);

return 0;

}

3) static\_cast is more restrictive than c-style.

Example: char\* to int\* is allowed in c-style but not with static\_cast.

#include <iostream>

using namespace std;

int main()

{

char c = 'c';

int \*p = (int\*)&c;// PASS compile time but fail at runtime or gives undefined behavior

cout<<\*p<<endl;

int \*p1 = static\_cast<int\*>(&c);//FAILS at compile time

return 0;

}

4) static\_cast avoids cast from derived to private base pointer.

Example:

#include <iostream>

using namespace std;

class Base

{

int x;

public:

Base()

{

cout<<"Inside the base: "<<endl;

}

private:

void print()

{

cout<<"Inside the private: "<<endl;

}

};

class Derived : private Base

{

public:

Derived()

{

cout<<"inside the derived: "<<endl;

}

};

int main()

{

Derived d;

Base \*p = (Base\*)&d;//allowed at compile time

Base \*p1 = static\_case<Base\*>(&d); // Not allowed at compile time

return 0;

}

5) static\_cast should be preferred when converting to void\* or from void\*

int main()

{

int i = 10;

void \*v = static\_case<void\*>(&i);

int \*ip = static\_cast<int\*>(v);

}

Disadvantages of static\_cast:

1) use for all upcasts, but never use for confusing downcast.

Example:

#include <iostream>

using namespace std;

class Base

{

};

class Derived1 : public Base

{

};

class Derived2 : public Base

{

};

int main()

{

Derived1 d1;

Derived2 d2;

Base \*bp1 = static\_cast<Base\*>(&d1);

Base \*bp2 = static\_cast<Base\*>(&d2);

Derived1 \*d1p = static\_cast<Derived1\*>(bp2);

Derived2 \*d2p = static\_cast<Derived2\*>(bp1);;

return 0;

}

10) const\_cast in c++?

The expression const\_cast<T>(v) can be used to change the const or volatile qualifiers of pointers or references.

**where T must be a pointer, reference, or pointer to member type.**

1) When the actual referred object/variable is not const.

#include <iostream>

using namespace std;

int main()

{

const int a = 10;

const int \*b1 = &a;

int \*d1 = const\_cast<int\*>(b1);

\*d1 = 15; // invalid and undefined behavior

int a2 = 20;

const int \*b2 = &a2;

int \*d2 = const\_cast<int\*>(b2);

\*d2 = 30; // valid code

return 0;

}

2) When we need to call some 3rd party library where it is taking variable/object as non-const but not changing that.

void thirdpartylibrary(int \*x)

{

int k = 10;

cout<<k+\*(x);

}

int main()

{

const int x = 20;

const int \*px = &x;

thirdpartylibrary(const\_cast<int\*>(px));

return 0;

}

11) reinterpret\_cast in c++?

It can perform dangerous conversions because it can typecast any pointer to any other pointer.

Example:

class Mango

{

public:

void eatmango()

{

cout<<"eating mango" <<endl;

}

};

class Banana

{

public:

void eatBanana()

{

cout<<"eating banana"<<endl;

}

};

int main()

{

Banana \*b = new Banana();

Manfo \*m = new Mango();

Banana \*newbanana = reinterpret\_cast<Banana\*>(m);

newbanana->eatBanana();

return 0;

}

output : eating banana.

2) It is used when you want to work with bits.

Example:

struct mystruct

{

int x;

int y;

char c;

bool b;

};

int main()

{

mystruct s;

s.x = 5;

s.y = 10;

s.c = 'a';

s.b = true;

int \*p = reinterpret\_cast<int\*>(&s);

cout<<\*p<<endl;

return 0;

}

Bottom Line:

The result of reinterpret\_cast cannot safely be used for anything other than being cast back to its original type.

we should be very careful when using this cast.

If we use this type of cast, then it becomes a non-portable product.

12) why do we use dynamic\_cast in c++?

i) dynamic\_cast is used at run time to find out the correct down-cast.

Note 1 : Need at least one virtual function in base class.

Note 2 : If the cast is successful, dynamic\_cast returns a value of type new\_type.

Note 3 : If the cast fails and new type is a reference type, it Throws an exception that matches a handler of type std::bad\_cast.

#include <iostream>

#include <exception>

using namespace std;

class Base

{

virtual void print()

{

cout<<"Inside the base class : "<<endl;

}

};

class derived1 : public Base

{

void print()

{

cout<<"Inside the derived1 class : "<<endl;

}

};

class derived2 : public Base

{

void print()

{

cout<<"Inside derived2 class : "<<endl;

}

};

int main()

{

derived1 d1;

Base \*bp = dynamic\_cast<Base\*>(&d1);

derived2 \*d2 = dynamic\_cast<derived2\*>(bp);

if(d2 == nullptr)

cout<<"NULL"<<endl;

else

cout<<"Not NULL"<<endl;

///for reference compiler return std::bad\_cast exception

try

{

derived2 &dr2 = dynamic\_cast<derived2&>(d1);

}

catch(std::exception &e)

{

cout<<e.what()<<endl;

}

return 0;

}

13) When do we reference over pointer and vice versa?

REFERENCE: use references in function parameters and return type.

1) Pass big objects

2) To avoid object slicing

3) To modify the local variable of the caller function

4) To achieve runtime polymorphism in a function.

POINTER: use pointers in algorithms and data structures like linked list, tree , graph etc.

Reason 1 : sometimes we put a NULL/null pointer in node

Reason 2 : sometimes we change the pointer to point some other node.

**14) What are Copy Elision, RVO, and NRVO?**

Copy Elision is an Optimization technique in which the compiler ignores the class’s copy and moves constructors. In such cases, the compiler code itself creates a copy of the temporary object.

Example:

Creating a new object with a temporary object.

MainFunda obj = MainFunda();

Return Value Optimization (RVO): RVO is a special kind of copy elision. The compiler may apply RVO when a function returns a temporary object.

15) What is code bloating in c++?

Code bloat is the production of code that is perceived as unnecessarily long, slow, or otherwise wasteful of resources.

#include <iostream>

using namespace std;

int main()

{

std::string str = "Hi, I am in main ";

// Here we are just printing the string for that no need to create a string variable

// directly we can print cout<<"Hi, I am in main" <<endl; like this.

// code bloating is nothing but if create a variable or object unnesserily that cause the code bloating

cout<<str<<endl;

return 0;

}

16) what is a diamond problem in c++?

In the below program if we remove the virtual keyword from class B and class C derived time then Class B and C derived from class A and Class D is derived from class B and Class C now Class D contains 2 copies of class A information that means compiler creates 2 times memory for class A in Class D. If you try to access class A member variables by using class D object then compiler don't understand which copy need to use and it gives the ambiguity exception it is called diamond problem to overcome this use the virtual keyword when deriving from class B and Class C from class A.

#include <iostream>

using namespace std;

class A

{

double a;

};

class B: virtual public A

{

};

class C: virtual public A

{

};

class D: public B,public C

{

};

int main()

{

// without virtual keyword used in B and C

cout<<"Class A size : "<<sizeof(A)<<endl; // size of class A is 4 bytes

cout<<"Class B size : "<<sizeof(B)<<endl; // size of class B is 4 bytes because inherited from A

cout<<"Class C size : "<<sizeof(C)<<endl; // size of class C is 4 bytes because inherited from A

cout<<"Class D size : "<<sizeof(D)<<endl; // size of class D is 8 bytes because inherited from B and C so compiler creates 2 copies of class A

return 0;

}

17) what is placement new in c++?

#include <iostream>

using namespace std;

class Base

{

public:

Base(){cout<<"constructor : "<<endl;}

~Base(){ cout<<"Destructor : "<<endl;}

};

int main()

{

//Normal case:

cout<<"Normal case : "<<endl;

Base \*obj = new Base();

delete obj;

//Placement new case

std::cout<<"Placement new case : "<<std::endl;

char \*memory = new char[10\*sizeof(Base)];

Base \*obj1 = new(&memory[0]) Base();

Base \*obj2 = new(&memory[1]) Base();

obj1->~Base();

obj2->~Base();

delete[] memory;

return 1;

};

https://godbolt.org/ this website tells which code is faster and optmizations (https://www.youtube.com/watch?v=\_fzAF1WAlgM&list=PLk6CEY9XxSIDy8qVHZV-Nf-r9f2BkRZ6p&index=65)

17) Difference between operator [] and at() to access vector in c++?

* Both are used to access elements in vectors.
* operator[] doesn’t do range checking, at() does range checking before accessing
* operator[] doesn’t throw when it is out of bound (undefined behavior), but at() throw if it is out of bound.
* operator[] faster and at() is slower as compare to operator[]

Example:

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> vec = {3,5,6,7};

cout<<vec[0]<<endl; // output : 3

cout<<vec.at(0)<<endl; // output : 3

cout<<vec[100]<<endl; // output : undefined behaviour

cout<<vec.at(100)<<endl; // output : throws out of bound exception because it checks the range before accessing

return 0;

}

18) what are Rules 3 and Rule of 5?

RULE OF THREE:

1. Destructor

2. copy constructor

3.copy assignment operator

RULE OF FIVE :

1.Destructor

2.copy constructor

3.copy assignment operator

4.Move copy constructor

5.Move Assignment operator.

19) Advantages and disadvantages of the c++ program language?

* **Portability**: c++ provides this feature of portability allowing us to develop codes without caring about the hardware. This lets us move the development of the program from one platform to another.
  + For example, if you're working on windows OS and for some reason, you have to switch to Linux, the codes from windows OS will also run in the LINUX OS without any error.
* **Mid-level programing language**?
  + Being a mid-level programming language, we can treat it as both a low-level and high-level language. Features of high-level language help to develop games and desktop applications, whereas features of low-level language help make kernels.
* **Object Oriented**

The OOP concepts like polymorphism, encapsulation, inheritance, and abstraction give c++ the biggest advantage over other programming languages. It proved to be of great significance since this feature was not in c, this helped users to treat data as objects and classes.

* **Multi paradigm programing language:**

Paradigm refers to the planning involved in programming. It concerns the logic, the style, and the way how we proceed with the program. c++ is a multi-paradigm programing language as it follows three paradigms:

* + - Generic - Using a single idea that serves multiple purposes.
    - Imperative - Using steps that change the state of the program.
    - Object Oriented - using methods and classes for reusability and modularity.
* **Memory Management:** 
  + c++ supports DMA (Dynamic Memory Allocation), which helps to free and allocate memory. Since there is no garbage collection, c++ gives the programmer total control over memory management.
* **Fast and powerful :**
  + As c++ is a compiler-based programing language, we do not require to install a special runtime while running the program. hence, they are pre-interpreted and it making the code faster and more powerful.
* Similar to other languages:
  + c++ syntax is similar to c#, C, and Java.

**Disadvantage:**

1) Pointers

2) No garbage collection

3) Unsafe

4) Complex

5) No built-in threads

6) lack of algebraic data types

20) Association, Aggregation and composition?

**Association**: If two classes in a model need to communicate with each other, there must be a link

between them, and that can be represented by an association (connector).

**Aggregation**: Aggregation implies a relationship where the child can exist independently of the parent.

**Composition**: Composition implies a relationship where the child cannot exist independently of the

parent.

21) In p = new Fred(), does the Fred memory “leak” if the Fred constructor throws an exception?

No, If an exception occurs during the Fred of p = new Fred(). The c++ language guarantees that the memory sizeof(Fred) bytes that are allocated will be deallocated automatically and release back to the heap.

Here are the details: new Fred() is a two-step process:

* sizeof(Fred) bytes of memory are allocated using the primitive void\* operator new(size\_t nbytes).

This primitive is similar in spirit to malloc(size\_t bytes). (Note, however, that these two are not interchangeable; e.g., there is no guarantee that the two memory allocation primitives even use the same heap!).

* It constructs an object in that memory by calling the Fred constructor. The pointer returned from the first step is passed as this parameter to the constructor. This step is wrapped in a try … catch block to handle the case when an exception is thrown during this step.

Fred\* p;

void\* tmp = operator new(sizeof(Fred));

try

{

new(temp) Fred(); //placement new

p = (Fred\*) tmp; the pointer is assigned only if the ctor succeed

}

catch(...)

{

operator delete(tmp); //Deallocate the memory

throw; // Re-throw the exception

}

22) what are stack and heap?

The memory that the program uses is typically divided into a few different areas, called segments.

* The code segment (also called a text segment), is where the compiled program sits in memory. The code segment is typically read-only.
* The bss segment (also called the uninitialized data segment), where zero-initialized global and static variables are stored.
* The data segment (also called the initialized data segment), is where initialized global and static variables are stored.
* The heap, where dynamically allocated variables are allocated from.
* The call stack, where function parameters, local variables, and other function-related information are stored.

23) why do we need brace initialization?

Brace initialization was introduced to provide a more consistent initialization syntax.

Brace initialization has added benefit: It disallows "narrowing conversions". This means that if you try to brace initialize a variable using a value that the variable cannot safely hold, the compiler will produce an error.

#include <iostream>

using namespace std;

int main()

{

int x1 = 4.5; // output : 4

int x2;

x2 = 4.5; // output : 4

int x3 (4.5); // output : 4

int x4 = (4.5);// output : 4

int x5 {4.5}; //error: narrowing conversion of ‘4.5e+0’ from ‘double’ to ‘int’ [-Wnarrowing]

cout<<x1<<" "<<x2<<" "<<x3<<" "<<x4<<" "<<endl;

return 0;

}

24) What is the difference between initialization and assignment?

Initialization gives a variable an initial value at the point when it is created. The assignment gives a variable a value at some point after the variable is created.

25) What form of initialization should you be using?

Direct brace initialization.

27) Compile-time constants, constant expressions, and constexpr?

A constant expression is an expression that can be evaluated by the compiler at compile-time. To be a constant expression, all the values in the expression must be known at compile-time.

**Compile-time const**: A const variable is a compile-time constant if its initializer is a constant expression.

**Runtime const**: Any const variable that is initialized with a non-constant expression is a runtime constant. Runtime constants are constants whose initialization values aren’t known until runtime.

**constexpr:** we use the constexpr keyword instead of const in a variable’s declaration. A constexpr (which is short for “constant expression”) variable can only be a compile-time constant. If the initialization value of a constexpr variable is not a constant expression, the compiler will error.

#include <iostream>

int getNumber()

{

std::cout << "Enter a number: ";

int y{};

std::cin >> y;

return y;

}

int main()

{

const int x{ 3 }; // x is a compile time constant

const int y{ getNumber() }; // y is a runtime constant

const int z{ x + y }; // x + y is a runtime expression

std::cout << z << '\n'; // this is also a runtime expression

constexpr double gravity { 9.8 }; // ok: 9.8 is a constant expression

constexpr int sum { 4 + 5 }; // ok: 4 + 5 is a constant expression

constexpr int something { sum }; // ok: sum is a constant expression

constexpr int myAge { age }; // compile error: age is not a constant expression

constexpr int f { getNumber() }; // compile error: return value of five() is not a constant expression

return 0;

}

**Best Practice:**

Any variable that should not be modifiable after initialization and whose initializer is known at compile-time should be declared as constexpr.

Any variable that should not be modifiable after initialization and whose initializer is not known at compile-time should be declared as const.

**disadvantage**: Evaluating constant expressions at compile-time makes our compilation take longer (because the compiler must do more work), but such expressions only need to be evaluated once (rather than every time the program is run). The resulting executables are faster and use less memory.

28) string and string\_view?

**std::String**: Also note that std::string::length() returns an unsigned integral value (most likely of type size\_t). If you want to assign the length to an int variable, you should static\_cast it to avoid compiler warnings about signed/unsigned conversions:

int length { static\_cast<int>(name.length()) };

In C++20, you can also use the std::ssize() function to get the length of a std::string as a signed integer:

#include <iostream>

#include <string>

int main()

{

std::string name{ "Alex" };

std::cout << name << " has " << std::ssize(name) << " characters\n";

return 0;

}

Literals for std::string Double-quoted string literals (like “Hello, world!”) are C-style strings by default (and

thus, have a strange type).

We can create string literals with the type std::string by using a s suffix after the double-quoted string literal.

#include <iostream>

#include <string> // for std::string

#include <string\_view> // for std::string\_view

int main()

{

using namespace std::literals; // easiest way to access the s and sv suffixes

std::cout << "foo\n"; // no suffix is a C-style string literal

std::cout << "goo\n"s; // s suffix is a std::string literal

std::cout << "moo\n"sv; // sv suffix is a std::string\_view literal

return 0;

};

string\_view : std::string\_view C++17 To address the issue with std::string being expensive to initialize (or copy), C++17 introduced std::string\_view (which lives in the <string\_view> header). std::string\_view provides read-only access to an existing string (a C-style string literal, a std::string, or a char array) without making a copy.

The following example is identical to the prior one, except we’ve replaced std::string with std::string\_view.

**Advantage:**

Prefer std::string\_view over std::string when you need a read-only string, especially for function parameters.

constexpr std::string\_view

Example :

constexpr std::string\_view s{ "Hello, world!" };

std::cout << s << '\n'; // s will be replaced with "Hello, world!" at compile-time

Converting a std::string to a std::string\_view: A std::string\_view can be created using a std::string initializer, and a std::string will implicitly convert to a std::string\_view:

Example:

std::string s{ "Hello, world" };

std::string\_view sv{ s }; // Initialize a std::string\_view from a std::string

Converting a std::string\_view to a std::string : Because std::string makes a copy of its initializer, C++ won’t allow implicit conversion of a std::string from a std::string\_view. However, we can explicitly create a std::string with a std::string\_view initializer, or we can convert an existing std::string\_view to a std::string using static\_cast:

#include <iostream>

#include <string>

#include <string\_view>

void printString(std::string str)

{

std::cout << str << '\n';

}

int main()

{

std::string\_view sv{ "balloon" };

std::string str{ sv }; // okay, we can create std::string using std::string\_view initializer

// printString(sv); // compile error: won't implicitly convert std::string\_view to a std::string

printString(static\_cast<std::string>(sv)); // okay, we can explicitly cast a std::string\_view to a

//std::string

return 0;

}

29) code coverage?

The term code coverage is used to describe how much of the source code of a program is executed while testing. There are many different metrics used for code coverage. we'll cover a few of the more useful and popular ones in the following section.

**statement coverage:** The term statement coverage refers to the percentage of statements in your code that have been exercised by your testing routines.

30) what is the fetal error?

If the error is so bad that the program cannot continue to operate properly, this is called a non-recoverable error also called a fatal error. in such a case, the best thing to do is terminate the program.

31) what is Assertion?

Using a conditional statement to detect an invalid parameter along with printing an error message and terminating the program.

An asserting is an expression that will be true unless there is a bug in the program. If the expression evaluates to true, the assertion statement does nothing. If the conditional expression evaluates to false, an error message is displayed, and the program is terminated.

#include <cassert> // for assert()

#include <cmath> // for std::sqrt

#include <iostream>

double calculateTimeUntilObjectHitsGround(double initialHeight, double gravity)

{

assert(gravity > 0.0); // The object won't reach the ground unless there is positive gravity.

if (initialHeight <= 0.0)

{

// The object is already on the ground. Or buried.

return 0.0;

}

return std::sqrt((2.0 \* initialHeight) / gravity);

}

int main()

{

std::cout << "Took " << calculateTimeUntilObjectHitsGround(100.0, -9.8) << " second(s)\n";

return 0;

}

**Note**: The assert macro comes with a small performance cost that is incurred each time the assert condition is checked.

**Note1**: assert should never be encountered in production code. Consequently, many developers prefer that asserts are only active in debug builds. c++ comes with a way to turn off asserts in production code. If the macro NDEBUG is defined, the assert macro gets disabled.

**Some assert limitations and warnings**:

There are a few pitfalls and limitations to assert. First, the assert itself can have a bug. If this happens, the assert will either report an error where none exists or fail to report a bug where one does exist.

Second, your asserts should have no side effects -- that is, the program should run the same with and without the assert. Otherwise, what you are testing in a debug configuration will not be the same as in a release configuration (assuming you ship with NDEBUG).

Also note that the abort() function terminates the program immediately, without a chance to do any further cleanup (e.g. close a file or database). Because of this, asserts should be used only in cases where corruption isn’t likely to occur if the program terminates unexpectedly.

32**) what is static\_assert?**

c++ also has another type of assert called static\_assert. A static\_assert is an assertion that is checked at compile-time rather than at runtime, with a falling static\_assert causing a compile error. unlike assert, which is declared in the <cassert> header, static\_assert is a keyword, so no header needs to be included to use it.

33) **Null pointer?**

When a pointer is holding a null value, it means the pointer is not pointing at anything. Such a pointer is called a null pointer.

example: int\* ptr {}; // ptr is a null pointer, and is not holding an address

34) **The nullptr keyword?**

Much like the keywords true and false represent Boolean literal values, The nullptr keyword represents a null pointer literal. we can use nullptr to explicitly initialize or assign a pointer a null value.

int\* ptr { nullptr }; // can use nullptr to initialize a pointer to be a null pointer

35) **What are Covariant return types?**

There is one special case in which a derived class virtual function override can have a different return type than the base class and still be considered a matching override. If the return type of a virtual function is a pointer or a reference to some class, the override function can return a pointer or a reference to a derived class. These are called covariant return types.

36) **What is ADL?**

Argument Dependent Lookup is the set of rules for looking up unqualified function names in function-call expressions, including implicit function, calls to overloaded operators. These function names are looked up in the namespaces of their arguments in addition to the scopes and namespaces considered by the usual unqualified name lookup.

Example:

#include <iostream>

using namespace std;

namespace ns

{

struct s {};

void callS(struct s) { cout<<"inside Ns namespaces:";};

}

namespace ns2

{

struct s {};

void callS(struct s) { cout<<"inside Ns2 namespaces:";};

}

int main()

{

ns::s s1; // defined by using namespaces

callS(s1);// compiler check the argument type and then checks in that namespace so it is called namespace ns method

return 0;

}

37) All those templates and template specializations must slow down my program, right?

Wrong.

This is a quality of implementation issue so your results may vary. However, there is usually no slowdown at all. If anything, the template might affect the speed of compilation slightly, but once the types are resolved by the compiler at compile-time, it will typically generate code that is just as fast as with non-template functions, including inline-expanding appropriate functions.

38) Templates are overloading, right?

Yes and no,

Function templates participate in name resolution for overloaded functions, but the rules are different. For a template to be considered in overload resolution, the type must match exactly. If the types do not match exactly, the conversions are not considered and the template is simply dropped from the set of viable functions. that's what is known as "SFINAE" - "Substitution Failure Is Not An Error".

Example:

#include <iostream>

#include <typeinfo>

template<typename T>

void foo(T\* x)

{ std::cout << "foo<" << typeid(T).name() << ">(T\*)\n"; }

void foo(int x)

{ std::cout << "foo(int)\n"; }

void foo(double x)

{ std::cout << "foo(double)\n"; }

int main()

{

foo(42); // matches foo(int) exactly

foo(42.0); // matches foo(double) exactly

foo("abcdef"); // matches foo<T>(T\*) with T = char

return 0;

}

39) **std::allocator**?

Allocators are objects responsible for encapsulating memory management std::allocator is used when you want to separate allocation and do construction in two steps. it is also used when separate destruction and deallocation are done in two steps.

Example :

#include <iostream>

#include <memory>

using namespace std;

int main()

{

// allocator for integer values

allocator<int> myAllocator;

// allocate space for five ints

int\* arr = myAllocator.allocate(5);

// construct arr[0] and arr[3]

// myAllocator.construct(arr, 100); // no longer allowed in C++20

arr[0] = 100; // do this instead

arr[3] = 10;

cout << arr[3] << endl;

cout << arr[0] << endl;

// deallocate space for five ints

myAllocator.deallocate(arr, 5);

return 0;

}

**Advantage of using std::allocator**

* allocator is the memory allocator for the STL containers. This container can separate the memory allocation and de-allocation from the initialization and destruction of their elements. Therefore, a call of vec.reserve(n) of a vector vec allocates only memory for at least n elements. The constructor for each element will not be executed.
* allocator can be adjusted according to the container of your need, for example, vector where you only want to allocate occasionally.
* On the contrary, new doesn’t allow to have control over which constructors are called and simply constructs all objects at the same time. That’s an advantage of std:: allocator over new

40) **R-value references**?

c++ 11 adds a new type of reference called an r-value reference. An r-value reference is a reference that is designed to be initialized with an r-value. while an l-value reference is created using a single ampersand, an r-value reference is created using double ampersand.

Note: R value cannot be initialized with l value;

R-value references have two properties :

* R-value references extend the lifespan of the object they are initialized with to the lifespan of the r-value reference.
* non-const r-value references allow you to modify the r-value.

41) Exception safety guarantees?

An exception safety guarantee is a contractual guideline about how functions or classes will behave in the event an exception occurs. There are four levels of exception safety:

* **No guarantee:** There are no guarantees about what will happen if an exception is thrown( a class may be left in an unusable state).
* **Basic guarantee**: If an exception is thrown, no memory will be leaked and the object is still usable, but the program may be left in a modified state.
* **strong guarantee**: If an exception is thrown, no memory will be leaked and the program state will not be changed. This means the function either completely succeeds or has no side effects if it fails. this is easy if the failure happens before anything is modified in the first place, but can also be achieved by rolling back any changes so the program is returned to the pre-failure state.
* **No throw/No fail**: The function will always succeed or fail without throwing an exception.(nothrow)

42) **std::weak\_ptr?**

The std::weak\_ptr are not directly usable. It uses a std::weak\_ptr, you must first convert it into a std::shared\_ptr. then you can use the std::shared\_ptr. Then you can use the std::shaerd\_ptr.To convert a std::weak\_ptr into std::shared\_ptr, you can use the lock() member function

Example :

#include <iostream>

#include <memory>

std::weak\_ptr<int> gw;

void observe()

{

std::cout << "gw.use\_count() == " << gw.use\_count() << "; ";

// we have to make a copy of shared pointer before usage:

if (std::shared\_ptr<int> spt = gw.lock()) {

std::cout << "\*spt == " << \*spt << '\n';

}

else {

std::cout << "gw is expired\n";

}

}

int main()

{

{

auto sp = std::make\_shared<int>(42);

gw = sp;

observe();

}

observe();

}

43) what is std::move\_if\_noexcept ?

The move constructors exception problem : A move operation transfer ownership of a given resource from the source to the destination object. If the move operation is interrupted by an exception after the transfer of ownership occurs, then our source object will be left in a modified state. This isn't a problem if the source object is a temporary object and going to be discarded after the move anyway. but for non temporary object, we've now damaged the source object. To comply with the strong exception guarantee, we'd need to move the resource back to the source object, but if the move failed the first time, there's no guarantee the move back will succeed either.

How can we give move constructor the strong exception guarantee? It is simple enough to avoid throwing exceptions in the body of a move constructor, but a move constructor may invoke other constructors that are potentially throwing.

Example :

#include <iostream>

#include <utility> // For std::pair, std::make\_pair, std::move, std::move\_if\_noexcept

#include <stdexcept> // std::runtime\_error

class MoveClass

{

private:

int\* m\_resource{};

public:

MoveClass() = default;

MoveClass(int resource)

: m\_resource{ new int{ resource } }

{}

// Copy constructor

MoveClass(const MoveClass& that)

{

// deep copy

if (that.m\_resource != nullptr)

{

m\_resource = new int{ \*that.m\_resource };

}

}

// Move constructor

MoveClass(MoveClass&& that) noexcept

: m\_resource{ that.m\_resource }

{

that.m\_resource = nullptr;

}

~MoveClass()

{

std::cout << "destroying " << \*this << '\n';

delete m\_resource;

}

friend std::ostream& operator<<(std::ostream& out, const MoveClass& moveClass)

{

out << "MoveClass(";

if (moveClass.m\_resource == nullptr)

{

out << "empty";

}

else

{

out << \*moveClass.m\_resource;

}

out << ')';

return out;

}

};

class CopyClass

{

public:

bool m\_throw{};

CopyClass() = default;

// Copy constructor throws an exception when copying from a CopyClass object where its m\_throw is 'true'

CopyClass(const CopyClass& that)

: m\_throw{ that.m\_throw }

{

if (m\_throw)

{

throw std::runtime\_error{ "abort!" };

}

}

};

int main()

{

// We can make a std::pair without any problems:

std::pair my\_pair{ MoveClass{ 13 }, CopyClass{} };

std::cout << "my\_pair.first: " << my\_pair.first << '\n';

// But the problem arises when we try to move that pair into another pair.

try

{

my\_pair.second.m\_throw = true; // To trigger copy constructor exception

// The following line will throw an exception

std::pair moved\_pair{ std::move\_if\_noexcept(my\_pair) }; // We'll comment out this line later

// std::pair moved\_pair{ std::move\_if\_noexcept(my\_pair) }; // We'll uncomment this line later

std::cout << "moved pair exists\n"; // Never prints

}

catch (const std::exception& ex)

{

std::cerr << "Error found: " << ex.what() << '\n';

}

std::cout << "my\_pair.first: " << my\_pair.first << '\n';

return 0;

}

noexcept functions are no-throw/no-fail, they implicitly meet the criteria for the strong exception guarantee. Thus, a noexcept move constructor is guaranteed to succeed.

we can use the standard library function std::move\_if\_noexcept() to determine whether a move or a copy should be performed. std::move\_if\_noexcept is a counterpart to std::move, and is used in the same way

If the compiler can tell that an object passed as an argument to std::move\_if\_noexcept won’t throw an exception when it is move constructed (or if the object is move-only and has no copy constructor), then std::move\_if\_noexcept will perform identically to std::move() (and return the object converted to an r-value). Otherwise, std::move\_if\_noexcept will return a normal l-value reference to the object.

44) Template Argument deduction?

We can use the template argument deduction to have the compiler deduce the actual type that should be used from the argument types in the function call.

Example:

max<int>(1,2);

we can do one of these instead

max<>(1,2); // the compiler will only consider max<int> template function overloads when determining which overloaded function to call.

max(1,2); //the compiler will consider both max<int> template function overloads and max non-template function overloads.

Example:

#include <iostream>

template <typename T>

T max(T x, T y)

{

std::cout << "called max<int>(int, int)\n";

return (x > y) ? x : y;

}

int max(int x, int y)

{

std::cout << "called max(int, int)\n";

return (x > y) ? x : y;

}

int main()

{

std::cout << max<int>(1, 2) << '\n'; // selects max<int>(int, int)

std::cout << max<>(1, 2) << '\n'; // deduces max<int>(int, int) (non-template functions not considered)

std::cout << max(1, 2) << '\n'; // calls function max(int, int)

return 0;

}

45) what are the disadvantages of templates?

* First, the compiler will create (and compile) a function for each function call with a unique set of argument types. So while function templates are compact to write, they can expand into a crazy amount of code, which can lead to code bloat and slow compile times.
* The bigger downside of function templates is that they tend to produce crazy-looking, borderline unreadable error messages that are much harder to decipher than those of regular functions.

46) what are Abbreviated function templates?

c++ 20 introduces a new use of the auto keyword: when the auto keyword is used as a parameter type in a normal function, the compiler will automatically convert the function into a function template with each auto parameter becoming an independent template type parameter. this method for creating a function template is called an abbreviated function template.

template<typename T, typename U>

auto max(T x, U y)

{

return (x > y )?x:y;

}

instead

auto max(auto x, auto y)

{

return (x > y )?x:y;

}

47) what is std::tuple and why we need it?

std::tuple is a type that can bind fixed sizes of heterogeneous values together. We need to specify the type of elements as template parameters while creating a tuple object.

**creating tuple object**:

header : #include <tuple>

std::tuple<int, char, duble> result {7, 'c', 2.5};

Let get the first element from tuple.

int val = std::get<0>(result);

Getting outof Range value from Tuple:

Fetching any element with index more than number of elements encapsulated by tuple will cause compile time error.

int val2 = std::get<4>(result); //compile time error

Wrong type casting while getting value from tuple

std::string strval = std::get<0>(result); //compile time error.

Getting value from dynamic index:

int x = 1;

double dval = std::get<x>(result); //compiletime error

48) what is std::make\_tuple ?

std::make\_tuple creates a std::tuple object by deducing the target types of elements in tuple from the types of arguments.

auto result = std::make\_tuple(7, 9.8, "text");

50) Inline function in c++?

inlining is only a request to the compiler, not a command. compiler can ignore the request for inlining. compiler may not perform inlining in such circumstances like.

i) If a function contains a loop.

ii) If a function contains static variable.

iii) If a function is recursive.

iv) If the function return type is other than void, and the return statement doesn't exist in the function

body.

Advantages:

i) Function call overhead doesn't occur.

ii) It also saves the

**51) when should you use unsigned numbers?**

1. Unsigned numbers are preferred when dealing with bit manipulation. They are also useful when well-defined wrap-around behavior is required (like encryption and random number generation)
2. Unsigned numbers is still unavoidable in some cases, the unsigned value can be converted to signed value.

52) What Enumerations?

An Enumeration is a compound data type where every possible value is defined as a symbolic constant.

C++ supports two kinds of enumerations: unscoped enumerations and scoped enumerations.

53) What types of storage classes are available in c++?

Storage Classes are used to describe the features of a variable/function. These features basically include the scope. These features basically include the scope, visibility, and life-time which help us to trace the existence of a particular variable during the runtime of a program.

Auto

Register

Extern

Static: They are initialized only once and exist until the termination of the program. Thus, no new memory is

allocated because they are not re-declared.

Mutable

Chart, table

Description automatically generated

54) **what is Lambda Expression?**

A lambda expression allows us to define an anonymous function inside another function.

Lambdas aren’t functions. They are a special kind of object called a functor. Functors are objects that

contain an overloaded operator() that makes them callable like a function.

Syntax: [ captureClause ] (parameters) -> returnType

{

Statements;

}

Note:

* The capture clause can be empty if no captures are needed.
* The parameter list can be either empty or omitted if no parameters are required.
* The return type is optional, and if omitted, the auto will be assumed. While we previously noted that type deduction for function return types should be avoided.

**Generic lambdas and static variables:**

One thing to be aware of is that a unique lambda will be generated for each different type that auto

resolves to.

#include <iostream>

#include <algorithm>

#include <array>

#include <string\_view>

using namespace std;

void print(int count, const auto& fn)

{

for(int i { 0 } ; i < count; i++)

{

fn(i);

}

}

int main()

{

auto print{

[](auto value){ static int callCount { 0 };

std::cout<<callCount++<<": "<<value<<endl;

}

};

print("hello");

print(1.0);

print(1);

return 0;

}

Note: Most of the time, this is inconsequential. However, note that if the generic lambda uses static

duration variables, those variables are not shared between the generated lambdas.

**Return type deduction and trailing return types:**

If return type deduction is used, a lambda’s return type is deduced from the return statements inside the lambda, all return statements in the lambda must return the same type.

If you’re trying to return the different types of return types from the same lambda function then it produces the compile error that will give the

**In the case where we’re returning different types, we have two options:**

* Do explicit casts to make all the return types match, or
* explicitly specify a return type for the lambda, and let the compiler do implicit conversions.

**Example**: In the below example if you uncomment the commented code then you will get the return

Type error.

#include <iostream>

int main()

{

/\*auto divide{ [](int x, int y, bool intDivision) { // note: no specified return type

if (intDivision)

return x / y; // return type is int

else

return static\_cast<double>(x) / y; // ERROR: return type doesn't match previous return type

} };\*/

// note: explicitly specifying this returns a double

auto divide{ [](int x, int y, bool intDivision) -> double {

if (intDivision)

return x / y; // will do an implicit conversion of result to double

else

return static\_cast<double>(x) / y;

} };

std::cout << divide(3, 2, true) << '\n';

std::cout << divide(3, 2, false) << '\n';

return 0;

}

55) what are lambda captures?

The capture clause is used to give lambda access to variables available in the surrounding scope that it normally would not have access to.

The captured variables of a lambda are clones of the outer scope variables, not the actual variables.

Captures default to const value.

We can mark the lambda as Mutable. The mutable keyword in this context removes the const qualification.

**Capture by reference**.

Standard library functions may copy function objects (reminder: lambdas are function objects). If you want to provide lambdas with mutable captured variables, pass them by reference using std::ref.

Try to avoid mutable lambdas. Non-mutable lambdas are easier to understand and don’t suffer from the above issues, as well as more dangerous issues that arise when you add parallel execution.

55) What is the Iterators?

An Iterator is an object to traverse through a container providing access to each element along the way.

**Pointers as an iterator**: The simplest kind of iterator is a pointer that works for data stored sequentially in

memory.

Text

Description automatically generated

**Range-based for loops:** All types that have both begin () and end() member functions or that can be used

with std::begin() and std::end(), are usable in range-based for-loops.

Text

Description automatically generated

Range-based for-loops aren’t the only thing that makes use of iterators. They’re also used in std::sort and other algorithms. Now that you know what they are, you’ll notice they’re used quite a bit in the standard library.

**Iterator invalidation (dangling iterators):-** Much like Pointers and references, Iterators can be left “dangling”. If the elements being iterated over change address or are destroyed. When this happens, we say the iterator has been invalidated. Accessing an invalidated iterator produces undefined behavior.

Graphical user interface, text, application

Description automatically generated

56) What are the standard Library Algorithms?

The functionality provided in the algorithms library generally falls into one of three categories.

Inspectors – Used to view data in a container. Example : searching and counting

Mutators - Used to modify data in the container. Example: sorting and shuffling.

Facilitators \_ Used to generate a result based on values of the data members. Example: multiply values.

Std::find : It searches for the first occurrence of a value in a container. It takes 3 parameters: an iterator to the starting element in the sequence, an iterator to the ending element in the sequence, and a value to search for. It returns an iterator pointing to the element or the end of the container.

Std::find\_if: Sometimes we want to see if there is a value in a container that matches some condition rather than an exact value. In such cases, std::find\_if is perfect. The std::find\_if functions works similarly to std::find, but instead of passing in a value to search for, we pass in a callable object, such as a function pointer that checks to see if a match is found.

**std::count and std::count\_if :** [std::count](https://en.cppreference.com/w/cpp/algorithm/count) and std::count\_if search for all occurrences of an element or an element fulfilling a condition.

57) **What is an RAII (Resource Acquisition Is Initialization)?**

The RAII is a programming technique whereby the resource used is tied to the lifetime of objects with automatic duration. In c++, RAII is implemented via classes with constructors and destructors. A resource is typically acquired in the object’s constructor. That resource can then be used while the object is alive. The source is released in the destructor when the object is destroyed.

RAII – allocation in the constructor, deallocation in the destructor.

The primary advantage of RAII is that it helps prevent resource leaks

58) What is the difference between std::span vs std::string\_view?

Std::span<T> is a general-purpose array view template, whereas the std::String\_view is a more specialized view on a char sequence or string.

1. Span is a template, but string\_view is not.
2. String\_view is a read-only view.
3. String\_view supports str::string like operations

59) what is cache hit and cache miss?

The user has a memory machine. It has one layer for data storage and another layer for the cache. The user has stored an array with length N in the first layer. When the CPU needs data, It immediately checks in cache memory whether it has data or not. If data is present it results in CACHE HITS, else CACHE MISS, I.e., data is not in cache memory, so it retrieves data from main memory and inserts a block of data into the cache layer.

60) what are DRY, KISS and YAGNI principles?

**DRY principle :** (Don't repeat yourself ): This means that you should not have duplicated code. It’s easier to maintain code that is only in one place because if you need to change something in the code, you just need to change it in one place. Besides that, if you have the same code in two or more places, the chance of this code becoming different during the time is high, and when this happens it will become an easy way to introduce bugs in your system. Duplicated code also makes the code more complex and unnecessarily larger.

**KISS - (Keep it simple and stupid):** This principle says to make your code simple. You should avoid unnecessary complexity. A simple code it’s easier to maintain and easier to understand.

**YAGNI : (You ain't gonna need it)** : This principle says that you should not create features that it’s not really necessary.

This principle’s similar to the KISS principle, as both of them aim for a simpler solution. The difference between them it’s that YAGNI focus on removing unnecessary functionality and logic, and KISS focuses on complexity.

61) what is the complexity of containers ?

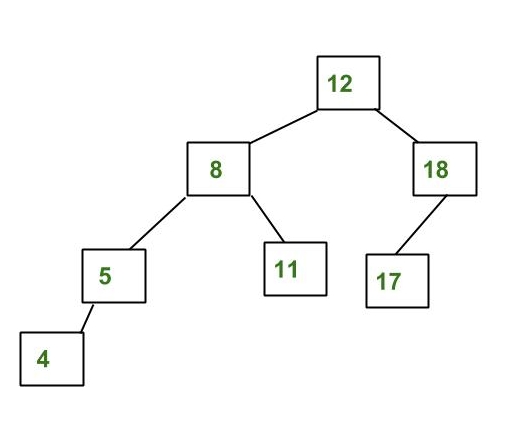
|  |  |  |  |
| --- | --- | --- | --- |
| 1. | Unordered Set | Hash Table (A hash table uses a hash function to compute an index) | For Insert, Search, Delete:  Best: O(1) Avg: O(1) Worst: O(N) |
| 2. | Set | Red Black Tree (self-balancing binary search tree – maintains the height of tree log N) | For Insert, Search, Delete: Best: O(log N) Avg: O(log N) Worst: O(log N) |
| 3. | Unordered Map | Hash Table (A hash table uses a hash function to compute an index) | For Insert, Search, Delete:  Best: O(1) Avg: O(1) Worst: O(N) |
| 4. | Map | Red Black Tree (self-balancing binary search tree – maintains the height of tree log N) | For Insert, Search, Delete: Best: O(log N) Avg: O(log N) Worst: O(log N) |
| 5. | Priority\_queue | Max Heap | Insert/Push: O(log N) Delete/Pop: O(log N) Peek/Top: O(1) |
| 6. | Stack | Linked List | Push: O(1) Pop: O(1) Top: O(1) |
| 7. | Queue | Linked List | Push: O(1) Pop: O(1) Front: O(1) Back: O(1) |

62) What is the AVL tree?

The AVL tree is a self-balancing binary search tree where the difference between the heights of left and right subtrees cannot be more than one for all nodes.

### Example of AVL Tree: The below tree is AVL because of the differences between the heights of the left

### and right subtrees for every node are less than or equal to 1.



### Example of a Tree that is NOT an AVL Tree: The below tree is not AVL because the differences

### between the heights of the left and right subtrees for 8 and 12 are greater than 1.

### Lightbox

**Time Complexity:** O(log(n)), For Insertion  
 **Auxiliary Space:** O(1)

63) What is the Red-Black tree?

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time

#### Rules That Every Red-Black Tree Follows:

* Every node has a color either red or black.
* The root of the tree is always black.
* There are no two adjacent red nodes (A red node cannot have a red parent or red child).
* Every path from a node (including root) to any of its descendant’s NULL nodes has the same number of black nodes.
* All leaf nodes are black nodes.

How does a Red-Black Tree ensure balance?  
A simple example to understand balancing is, that a chain of 3 nodes is not possible in the Red-Black tree. We can try any combination of colors and see if all of them violate the Red-Black tree property.

Chart, diagram, bubble chart

Description automatically generated

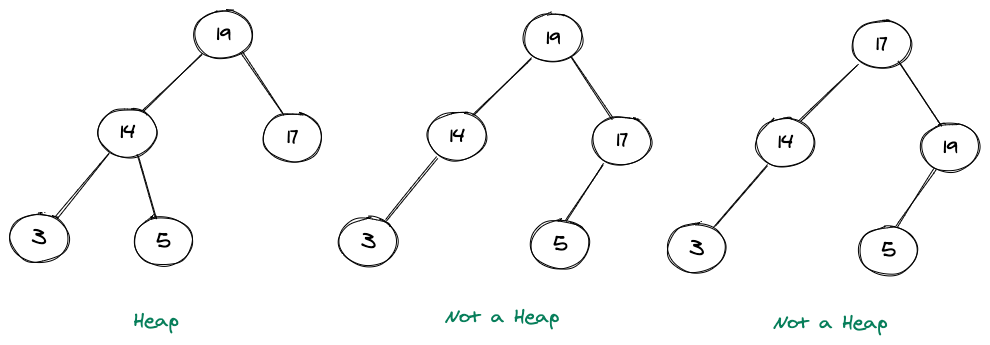
64) difference between the AVL tree and the Red-black tree?

* The AVL trees are more balanced compared to the Red-black trees and should be preferred but they may cause more rotation during the insertion and deletions.
* If your application involves many frequent insertions and deletions the Red-black tree should be preferred.
* If the insertions and deletions are less frequent and search is the more frequent operation the AVL tree should be preferred.

65) What is the Heap data structure?

A Heap is a special type of tree that follows two properties.

1. All leaves must be at h or h-1 levels
2. The value of the node must be >= or <= of its children nodes, known as the heap property.



In the pictures shown above, the leftmost tree denotes a heap (Max Heap) and the two tree to its right aren't heap as the middle tree violates the first heap property(not a complete binary tree) and the last tree from the left violates the second heap property(17 < 19).

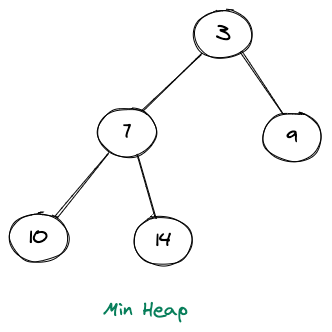
## **Types of Heap:** If we consider the properties of a heap, then we can have two types of heaps. These mainly are:

* Min Heap
* Max Heap

### Min Heap

In this heap, the value of a node must be less than or equal to the values of its children nodes.

Consider the pictorial representation of a Min Heap below:

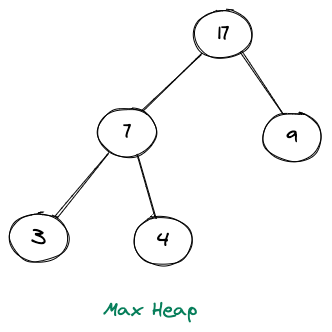


It can be clearly seen that the value of any node in the above heap is always less than the value of its children nodes.

### Max Heap

In this heap, the value of a node must be greater than or equal to the values of its children nodes.

Consider the pictorial representation of a Max Heap below:



It can be clearly seen that the value of any node in the above heap is always greater than the value of its children nodes.

## **Applications of Binary Heaps**

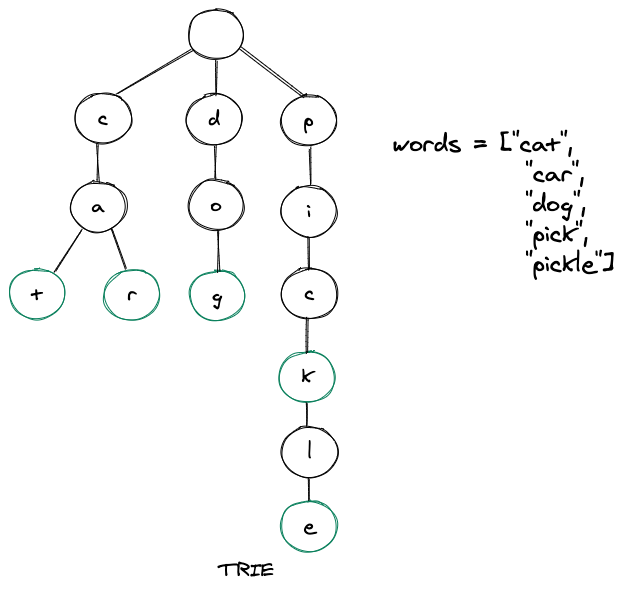
* Binary heaps are used in a famous sorting algorithm known as Heap sort.
* Binary heaps are also the main reason of implementing priority queues, as because of them the several priority queue operations like add(), remove() etc gets a time complexity of O(n).
* They are also the most preferred choice for solving Kth smallest / Kth Largest element questions.

**66) What is the trie data structure?**

A trie is an advanced data structure that is sometimes also known as a prefix tree or digital tree. It is a tree that stores the data in an ordered and efficient way. We generally use trie’s to store strings. Each node of a trie can have as many as 26 preferences

**Each node of a trie consists of two things**:

* A character
* A boolean value is used to implement whether this character represents the end of the word.



## **Why use Trie Data Structure?**

When we talk about the **fastest ways to retrieve values** from a data structure, **hash tables generally come to our minds**. Though very efficient in nature still very less talked about as when compared to hash tables, **trie's are much more efficient than hash tables** and also they possess several advantages over the same. Mainly:

* There won't be any collisions hence making the worst performance better than a hash table that is not implemented properly.
* No need for hash functions.
* Lookup time for a string in trie is O(k) where **k = length of the word**.
* It can take even less than O(k) time when the word is not there in a trie.

## **Trie Applications**

* The most common use of tries in the real world is the **autocomplete feature** that we get on a search engine (now everywhere else too). After we type something in the search bar, the tree of the potential words that we might enter is greatly reduced, which in turn allows the program to enumerate what kinds of strings are possible for the words we have typed in.
* Trie also helps in the case where we want to store additional information of in a word, say the popularity of the word, which makes it so powerful. You might have seen that when you type **"foot"**on the search bar, you get **"football"** before anything say **"footpath"**. It is because **"football"** is a much popular word.
* Trie also has helped in checking the correct spellings of a word, as the path is similar for a slightly misspelled word.
* **String matching** is another case where tries to excel a lot.

67) What is the B-tree?

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees, It assumed that everything is in the main memory.

To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in the main memory. When the number of keys is high, the data is read from the disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc ) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting the maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, etc.

## **Time Complexity of B-Tree:**

|  |  |  |
| --- | --- | --- |
| Sr. No. | Algorithm | Time Complexity |
| 1. | Search | O(log n) |
| 2. | Insert | O(log n) |
| 3. | Delete | O(log n) |

## **Properties of B-Tree:**

* All leaves are at the same level.
* B-Tree is defined by the term minimum degree ‘**t**‘. The value of ‘**t**‘ depends upon disk block size.
* Every node except the root must contain at least t-1 keys. The root may contain a minimum of **1** key.
* All nodes (including root) may contain at most (**2\*t – 1**) keys.
* Number of children of a node is equal to the number of keys in it plus **1**.
* All keys of a node are sorted in increasing order. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.
* B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink downward.
* Like other balanced Binary Search Trees, the time complexity to search, insert and delete is O(log n).
* Insertion of a Node in B-Tree happens only at Leaf Node.

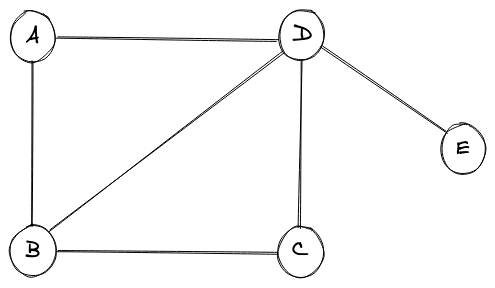
## **Applications of B-Trees**:

* It is used in large databases to access data stored on the disk
* Searching for data in a data set can be achieved in significantly less time using the B-Tree
* With the indexing feature, multilevel indexing can be achieved.
* Most of the servers also use the B-tree approach.

68) What is the Graph?

A **graph is an advanced data structure** that is used to organize items in an **interconnected network**. Each item in a graph is known as a **node**(or **vertex**) and these nodes are connected by **edges**.

In the figure below, we have a simple graph where there are five nodes in total and six edges.

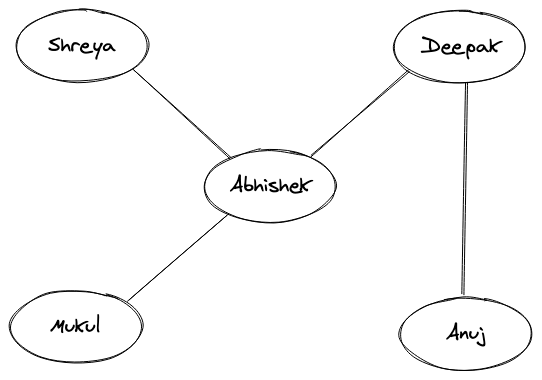


The nodes in any graph can be referred to as **entities** and the edges that connect different nodes define the **relationships between these entities**. In the above graph we have a set of nodes **{V} = {A, B, C, D, E}** and a set of edges, **{E} = {A-B, A-D, B-C, B-D, C-D, D-E}** respectively.

## Real-World Example

A very good example of graphs is a **network of socially connected people**, connected by a simple connection which is whether they know each other or not.

Consider the figure below, where a pictorial representation of a social network is shown, in which there are five people in total.



A line in the above representation between two people means that they know each other. If there's no line in between the names, then they simply don't know each other. The names here are equivalent to the nodes of a graph and the lines that define the relationship of "knowing each other" is simply the equivalent of an edge of a graph. It should also be noted that the relationship of knowing each other goes both ways like "Abhishek" knows "Mukul" and "Mukul" knows "Abhishek".

The social network depicted above is nothing but a graph.

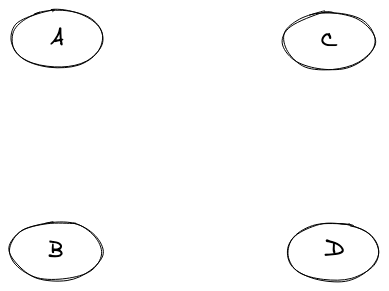
## Types of Graphs

Let's cover various different types of graphs.

### 1. Null Graphs

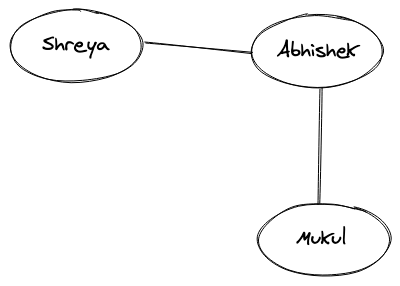
A graph is said to be null if there are no edges in that graph.

A pictorial representation of the null graph is given below:



### 2. Undirected Graphs

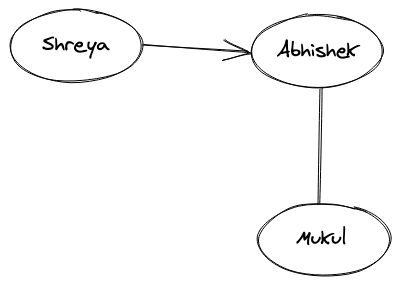
If we take a look at the pictorial representation that we had in the Real-world example above, we can clearly see that different node are connected by a link (i.e. edge) and that edge doesn't have any kind of direction associated with it. For example, the edge between "Anuj" and "Deepak" is bi-directional and hence the relationship between them is two ways, which turns out to be that "Anuj" knows "Deepak" and "Deepak" also knows about "Anuj". These types of graphs where the relation is bi-directional or there is not a single direction, are known as Undirected graphs.



### 3) Directed Graphs

What if the relation between the two people is something like, "Shreya" know "Abhishek" but "Abhishek" doesn't know "Shreya". This type of relationship is one-way, and it does include a direction. The edges with arrows basically denote the direction of the relationship and such graphs are known as directed graphs.

A pictorial representation of the graph is given below:

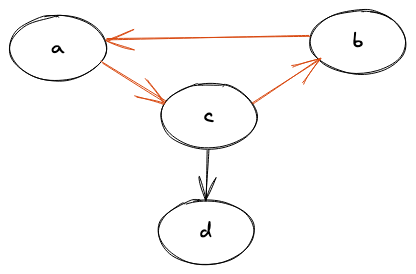


It can also be noted that the edge from "Shreya" to "Abhishek" is an outgoing edge for "Shreya" and an incoming edge for "Abhishek".

### 4. Cyclic Graph

A graph that contains at least one node that traverses back to itself is known as a cyclic graph. In simple words, a graph should have at least one cycle formation for it to be called a cyclic graph.

A pictorial representation of a cyclic graph is given below:

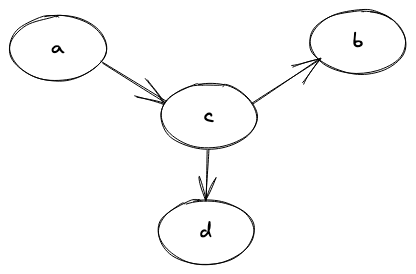


It can be easily seen that there exists a cycle between the nodes (a, b, c) and hence it is a cyclic graph.

### 5. Acyclic Graph

A graph where there's no way we can start from one node and can traverse back to the same one, or simply doesn't have a single cycle is known as an acyclic graph.

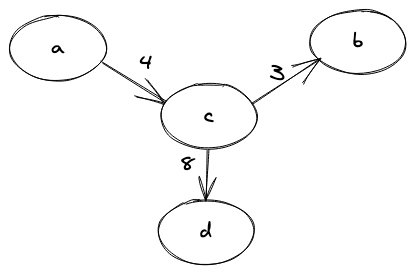
A pictorial representation of an acyclic graph is given below:



### 6) Weighted Graph

When the edge in a graph has some weight associated with it, we call that graph as a weighted graph. The weight is generally a number that could mean anything, totally dependent on the relationship between the nodes of that graph.

A pictorial representation of the weighted graph is given below:

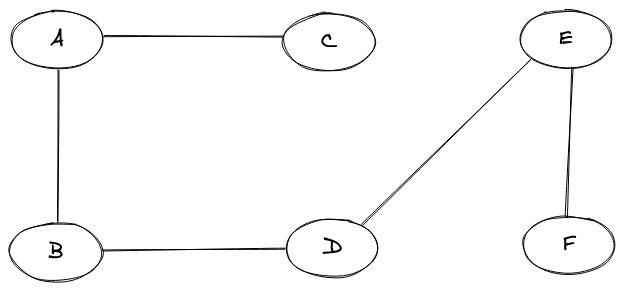


It can also be noted that if any graph doesn't have any weight associated with it, we simply call it an unweighted graph.

### 7. Connected Graph

A graph where we have a path between every two nodes of the graph is known as a connected graph. A path here means that we are able to traverse from a node "A" to say any node "B". In simple terms, we can say that if we start from one node of the graph we will always be able to traverse to all the other nodes of the graph from that node, hence the connectivity.

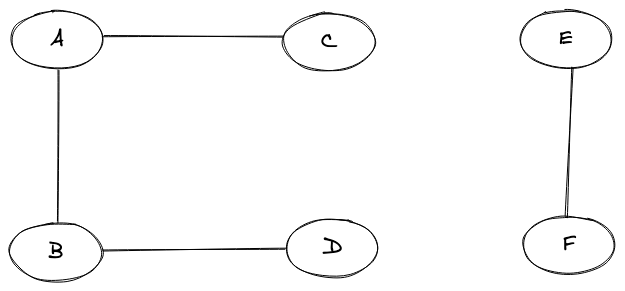
A pictorial representation of the connected graph is given below:



### 8) Disconnected Graph

A graph that is not connected is simply known as a disconnected graph. In a disconnected graph, we will not be able to find a path from between every two nodes of the graph.

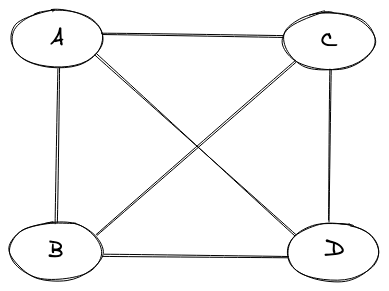
A pictorial representation of the disconnected graph is given below:



### 9. Complete Graph

A graph is said to be a complete graph if there exists an edge for every pair of vertices(nodes) of that graph.

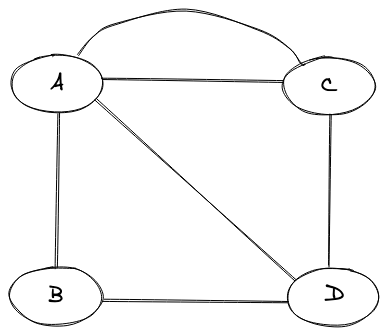
A pictorial representation of the complete graph is given below:



### 10. Multigraph

A graph is said to be a multigraph if there exist two or more than two edges between any pair of nodes in the graph.

A pictorial representation of the multigraph is given below:



## Commonly Used Graph Terminologies

* **Path** - A sequence of alternating nodes and edges such that each of the successive nodes are connected by the edge.
* **Cycle** - A path where the starting and the ending node is the same.
* **Simple Path** - A path where we do not encounter a vertex again.
* **Bridge** - An edge whose removal will simply make the graph disconnected.
* **Forest** - A forest is a graph without cycles.
* **Tree** - A connected graph that doesn't have any cycles.
* **Degree** - The degree in a graph is the number of edges that are incident on a particular node.
* **Neighbour** - We say vertex "A" and "B" are neighbors if there exists an edge between them.

69) What is the Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

**Types of**[**Recursions**](https://www.geeksforgeeks.org/recursion/)**:**   
Recursion are mainly of two types depending on whether a function calls itself from within itself or more than one function call one another mutually. The first one is called direct recursion and another one is called indirect recursion. Thus, the two types of recursion are:

1. **Direct Recursion**: These can be further categorized into four types:

* **Tail Recursion:** If a recursive function calling itself and that recursive call is the last statement in the function then it’s known as Tail Recursion. After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time.

Text

Description automatically generated

Output: 3 2 1

Let’s understand the example by **tracing tree of recursive function.** That is how the calls are made and how the outputs are produced.

A picture containing radar chart

Description automatically generated

**Time Complexity For Tail Recursion : O(n)**   
**Space Complexity For Tail Recursion : O(n)**  
**Note:** Time & Space Complexity is given for this specific example. It may vary for another example.

Let’s now converting Tail Recursion into Loop and compare each other in terms of Time & Space Complexity and decide which is more efficient.

Text

Description automatically generated

Output 3 2 1

**Time Complexity: O(n)   
Space Complexity: O(1)**

Note: Time & Space Complexity is given for this specific example. It may vary for another example.  
So it was seen that in case of loop the Space Complexity is O(1) so it was better to write code in loop instead of tail recursion in terms of Space Complexity which is more efficient than tail recursion.

**Why space complexity is less in case of loop ?**Before explaining this I am assuming that you are familiar with the knowledge that’s how the data stored in main memory during execution of a program. In brief,when the program executes,the main memory divided into three parts. One part for code section, the second one is heap memory and another one is stack memory. Remember that the program can directly access only the stack memory, it can’t directly access the heap memory so we need the help of pointer to access the heap memory.

Let’s now understand why space complexity is less in case of loop ?  
In case of loop when function “(void fun(int y))” executes there only one activation record created in stack memory(activation record created for only ‘y’ variable) so it takes only ‘one’ unit of memory inside stack so it’s space complexity is O(1) but in case of recursive function every time it calls itself for each call a separate activation record created in stack.So if there’s ‘n’ no of call then it takes ‘n’ unit of memory inside stack so it’s space complexity is O(n).

* **Head Recursion:** If a recursive function calling itself and that recursive call is the first statement in the function then it’s known as Head Recursion. There’s no statement, no operation before the call. The function doesn’t have to process or perform any operation at the time of calling and all operations are done at returning time.  
  Example:

Text

Description automatically generated

**Output** 1 2 3

Let’s understand the example by tracing tree of recursive function. That is how the calls are made and how the outputs are produced.

Diagram

Description automatically generated

**Time Complexity For Head Recursion: O(n)   
Space Complexity For Head Recursion: O(n)**

Note: Time & Space Complexity is given for this specific example. It may vary for another example.  
Note: Head recursion can’t easily convert into loop as Tail Recursion but it can. Let’s convert the above code into the loop.

Text

Description automatically generated

Output

1 2 3

* **Tree Recursion**: To understand Tree Recursion let’s first understand Linear Recursion. If a recursive function calling itself for one time then it’s known as Linear Recursion. Otherwise if a recursive function calling itself for more than one time then it’s known as Tree Recursion.  
  Example:  
  Pseudo Code for linear recursion

**fun(n)**

**{**

**// some code**

**if(n>0)**

**{**

**fun(n-1); // Calling itself only once**

**}**

**// some code**

**}**

Text

Description automatically generated

Output 3 2 1 1 2 1 1

Let’s understand the example by tracing tree of recursive function. That is how the calls are made and how the outputs are produced.

Chart, radar chart

Description automatically generated

**Time Complexity For Tree Recursion: O(2^n)   
Space Complexity For Tree Recursion: O(n)**Note: Time & Space Complexity is given for this specific example. It may vary for another example.

* **Nested Recursion:** In this recursion, a recursive function will pass the parameter as a recursive call. That means “recursion inside recursion”. Let see the example to understand this recursion.  
  Example:

Text

Description automatically generated

Output 91

Let’s understand the example by tracing tree of recursive function. That is how the calls are made and how the outputs are produced.

Diagram

Description automatically generated

**2.Indirect Recursion:** In this recursion, there may be more than one functions and they are calling one another in a circular manner.

![Diagram

Description automatically generated](data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEAYABgAAD/4RDmRXhpZgAATU0AKgAAAAgABAE7AAIAAAAJAAAISodpAAQAAAABAAAIVJydAAEAAAASAAAQzOocAAcAAAgMAAAAPgAAAAAc6gAAAAgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEFtaXlhIFBDAAAABZADAAIAAAAUAAAQopAEAAIAAAAUAAAQtpKRAAIAAAADMzYAAJKSAAIAAAADMzYAAOocAAcAAAgMAAAIlgAAAAAc6gAAAAgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADIwMTk6MDY6MDggMjM6MTg6MTMAMjAxOTowNjowOCAyMzoxODoxMwAAAEEAbQBpAHkAYQAgAFAAQwAAAP/hCxtodHRwOi8vbnMuYWRvYmUuY29tL3hhcC8xLjAvADw/eHBhY2tldCBiZWdpbj0n77u/JyBpZD0nVzVNME1wQ2VoaUh6cmVTek5UY3prYzlkJz8+DQo8eDp4bXBtZXRhIHhtbG5zOng9ImFkb2JlOm5zOm1ldGEvIj48cmRmOlJERiB4bWxuczpyZGY9Imh0dHA6Ly93d3cudzMub3JnLzE5OTkvMDIvMjItcmRmLXN5bnRheC1ucyMiPjxyZGY6RGVzY3JpcHRpb24gcmRmOmFib3V0PSJ1dWlkOmZhZjViZGQ1LWJhM2QtMTFkYS1hZDMxLWQzM2Q3NTE4MmYxYiIgeG1sbnM6ZGM9Imh0dHA6Ly9wdXJsLm9yZy9kYy9lbGVtZW50cy8xLjEvIi8+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczp4bXA9Imh0dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC8iPjx4bXA6Q3JlYXRlRGF0ZT4yMDE5LTA2LTA4VDIzOjE4OjEzLjM1NTwveG1wOkNyZWF0ZURhdGU+PC9yZGY6RGVzY3JpcHRpb24+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczpkYz0iaHR0cDovL3B1cmwub3JnL2RjL2VsZW1lbnRzLzEuMS8iPjxkYzpjcmVhdG9yPjxyZGY6U2VxIHhtbG5zOnJkZj0iaHR0cDovL3d3dy53My5vcmcvMTk5OS8wMi8yMi1yZGYtc3ludGF4LW5zIyI+PHJkZjpsaT5BbWl5YSBQQzwvcmRmOmxpPjwvcmRmOlNlcT4NCgkJCTwvZGM6Y3JlYXRvcj48L3JkZjpEZXNjcmlwdGlvbj48L3JkZjpSREY+PC94OnhtcG1ldGE+DQogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgIDw/eHBhY2tldCBlbmQ9J3cnPz7/2wBDAAcFBQYFBAcGBQYIBwcIChELCgkJChUPEAwRGBUaGRgVGBcbHichGx0lHRcYIi4iJSgpKywrGiAvMy8qMicqKyr/2wBDAQcICAoJChQLCxQqHBgcKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKir/wAARCADxAaUDASIAAhEBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtRAAAgEDAwIEAwUFBAQAAAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBkaEII0KxwRVS0fAkM2JyggkKFhcYGRolJicoKSo0NTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uHi4+Tl5ufo6erx8vP09fb3+Pn6/8QAHwEAAwEBAQEBAQEBAQAAAAAAAAECAwQFBgcICQoL/8QAtREAAgECBAQDBAcFBAQAAQJ3AAECAxEEBSExBhJBUQdhcRMiMoEIFEKRobHBCSMzUvAVYnLRChYkNOEl8RcYGRomJygpKjU2Nzg5OkNERUZHSElKU1RVVldYWVpjZGVmZ2hpanN0dXZ3eHl6goOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2Nna4uPk5ebn6Onq8vP09fb3+Pn6/9oADAMBAAIRAxEAPwD6RooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKK4T4w6X4y1fwE1t8PLqS31P7VG0ohuBBLJCM5VJDjad2xj8y5VWGTnafBP+ED/aG/5/PEH/hRp/8AH6APraivkDW/hB8S9Q8Nza38RPEdvY2mmbz/AMT3VpLho1O3LJsEg+Y4UKDuYgDHTPjdAH6N6hex6dptzfT/AOqtoXmf/dUEn+Vef/Cjxl4v8e2j6/q1no1n4fuPNWzit2lN0rK+BvJ+QjAbkYOQOAK0fGHhW2u/gzdaN4ilk1drDS8tcyu6PPNDFxK2Gzksu4gk9ec14/4J8M6RZfsr694gtrTZql/YXEVzceY58xUkYKNpO0YwOgFSpW55Pov8/wCvIpq6il1Z9L0V89eMZEH7GelguuWtrNV56kSLwPfg/lR49sjqXxH+FVj9rurRbmx8p5rSUxShWRQwDDkZGRx2Jq7e+4rvb8GzPm91Sfa/4n0LXmXw/wDF2veL/ih4x8y82+HdHmFja2oiQbpQcM+/G4/cJxnHzjjiuT8OaNZ/D39p1PDnhZZrTR9S0jzpbPz3dN4DEN8xJJ+TgnpuOOKu/s43Edv4R8VXN4yxvHrM0lwT/AAikkn8+1JNX5ulm/xt/mU7tcvmvyv/AJGt8Q/jjB4D8YLoi6G+pRRQxTX10lzs+yq77RldhycEHqOoFdt41n1T/hX+rXfhe8+z6jHZtcWkyRrJkqNwADAg7gMdO9fNGn+M/B2v+HPiJeeKtZFtrXiSVhZQyW00nlxx/NCNyoQBu2jr0UV7P8D/ABOnif4L2qTyB7jTYnsbgFuQEHyE/wDACv61Dv7KXdK/3/5aFaKou17fd/nqdH8LvFcnjT4baRrVy4e6li8u5YKFzKhKscDgZIzx611teOfsws5+FE6kkxpqcwjz0A2oePxJr2Otqnxaf1cyhe1n0uvuCiiioLCiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAqjrOs6d4e0a51bWruOzsbVN800h4UdO3JJJAAGSSQACTRrOs6d4e0a51bWruOzsbVN800h4UdO3JJJAAGSSQACTXyT4s8WeKP2g/HtvoPhu2kh0qJy9raO2EiUcG5uGGRnB99udq7mYlwA8WeLPFH7Qfj230Hw3bSQ6VE5e1tHbCRKODc3DDIzg++3O1dzMS+J8Z/AWnfDnxLpGiaZJJOTpMc9zcycGeYyyhn25IUYVQFHQAZJOSfrL4c/DnR/hv4bGm6UvnXMuHvb51xJcuO5/uqMnaucAE9SWY+Nftc/8AMo/9vv8A7QoA+iNTsItV0m70+4Z1iu4HgdkIDBWUqSM55wawvDngDRfDngP/AIRCIT32lMksci3jBmkWQksCVCj+I9AKn8A3M958NvDV1eTST3E2k2skssrlnkYwqSzE8kknJJroKVlr5ju9PI8n/wCGdfCb6LJpVzqniC5sxzaRTX4ZLEltzGFNm0FuhJDcH15rqtQ+G+kal4i8N6zPc3y3HhyMR2ipIgRxgD5wVyenYiuuop7O4nqcvP4B0uf4lW/jd57wanb2ptEiDr5JQhuSNu7PzH+L0rz3wf4MvrPxf8RfBuoWupWeg62TdWt/bRFY9smQ6LIVK7sOBj/ZNe1UUrLrtZr79fzD/NP7v+BoZHhXw1YeD/C9loOkeZ9ks0Ko0pBdskksxAAJJJPQVxN74Gtfhx4Y8aav4QXVry91iJ5FsIgJQszFgvlIibgAZPU8D2r02iia5736/qOL5bW6HD/BzwxN4S+FWj6dexPFdvGbi4jkXayPId20jsQCB+FdxRRVylzO5EVZWPnLxX+0l4j8K/EzVNHuvDVm2mafdPCIJGkjuJVAwsnmHKgNw4+Q/KwGT96up0H9p7wNqexNXTUNFl8kPI88HmxB+MorR7mPU4JRQQOcHAr1bVdF0rXrVbXXNMs9St0cSLFeW6zIrAEBgGBGcEjPua838Qfs4fD/AF26+0W9reaM7O7yDTZwqSFiD9xw6qBzgIFAzjHTElHomkeI9D8Qed/YOs6fqfkbfN+xXSTeXuzjdtJxnBxn0NaVfK+q/su+LNGulvfCfiCzvXtUE8TtvtLjzlJIWPG5QeFwxdeT2xmshvFvxy+GvmjVzrDWVpMk1xJqMH2yBt23CG5IbCn5VwsgwSQMNmgD6/or5y8M/tYQGNYvGPh6RHVGLXOlOGDtu+UCKQjaNvU7zyOnPHqXh/41fD/xFa+bb+JbOydURpIdScWroWBO358KxGCDsLAevIyAd3RRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAVR1nWdO8PaNc6trV3HZ2Nqm+aaQ8KOnbkkkgADJJIABJo1nWdO8PaNc6trV3HZ2Nqm+aaQ8KOnbkkkgADJJIABJr5J8WeLPFH7Qfj230Hw3bSQ6VE5e1tHbCRKODc3DDIzg++3O1dzMS4AeLPFnij9oPx7b6D4btpIdKicva2jthIlHBubhhkZwffbnau5mJf6X+HPw50f4b+GxpulL51zLh72+dcSXLjuf7qjJ2rnABPUlmJ8Ofhzo/w38NjTdKXzrmXD3t864kuXHc/3VGTtXOACepLMetoAK+cv2t7adrXwpdLDIbeN7uN5Qh2KzCIqpPQEhGIHfafQ19G0UAc/wCAbaez+G3hq1vIZILiHSbWOWKVCrxsIVBVgeQQRgg10FFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFAHJa98K/A3iXedX8Mae8skxnkngj+zyyOc5LSR7WbO4kgkgnk8gV5R4g/ZP0qS13eFfEN5b3CI58vUkWZJWwNg3IEKDOcnD9eBxg/QlFAHyB/wAIJ8afhRc7vDv9oTWizZU6PIbqCV3jwWNuQScAbdzx4BUYP3TW34W/ar1i08qDxfotvqMQ8pGurJvJlAHDuyHKux4IA8sZyOAePqSuf8TeBPC/jGNl8S6HZ37sixid49syqrbgqyrh1Gc8AjqfU0Ac34Z+O3w/8TRrs1yPS7jYztb6ri3KANt5cnyyTkEBXJwenBx6JXgHin9lTR7vzZ/CGtXGnSnzXW1vV86Ik8oiuMMijkEnzDjB5I586Wf4xfBBpLcC8GjWzq5Jj+1WDJ5rAYfB8oOxOQDG53AkA4oA+xKK8S8B/tMeHtf8mx8XRf2DqDbU+0ZL2sjfKM7useWLHDZVVXl69ptrmC8tYrqzmjnt5kEkUsThkkUjIZSOCCDkEUASUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAVR1nWdO8PaNc6trV3HZ2Nqm+aaQ8KOnbkkkgADJJIABJq1c3MFnay3V5NHBbwoZJZZXCpGoGSzE8AADJJr5E+IXi/WPjn8ULXwv4TuN+jibZp8Uo8lHKoTJcSZyTgByOMhOAoZmBAIvFnizxR+0H49t9B8N20kOlROXtbR2wkSjg3NwwyM4PvtztXczEv9L/AA5+HOj/AA38NjTdKXzrmXD3t864kuXHc/3VGTtXOACepLMT4c/DnR/hv4bGm6UvnXMuHvb51xJcuO5/uqMnaucAE9SWY9bQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQB5J44/Z08IeKt1zoyf8ACOagcfPZRAwN90fNBkAfKpxsKcsSd1eJW158Uf2fL+WF7bbpc82SJY/Psbl8OqsHGCjELu2go5CruGABX2RVHWdG07xDo1zpOtWkd5Y3SbJoZBww69RyCCAQRgggEEEUAcv8Ofit4e+JFgP7Lm+z6pHCJbvTZc+ZDztJBwBIucfMvQMu4KTiu2r5S+JvwQ1z4fX914u+Hl3cJpdtmXZbTut3YKwYOQw5aMDjdncFb5gQrOfTvgp8a4PHlqmh+IpI4PEkKcNgKl+oHLqOgcAZZB7svGQgB6/RRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUVxPxd8cf8ACAfDi+1WB9uoTYtLDjP79wcNyrD5VDPhhg7NvcUAeLftEfEufxDrP/Cu/DUMk6QXUa3clu5druf+GBVQncFZhkEE+YoGBsy3q3wb+FNj8OvDcVzcw7/EN9Cpvp5NpaHOCYEIJAVT1IJ3EZ6BQvmX7NXgK71PWbj4i69JJKQ80VmZ/MMk0zcS3BckBhhnT+LLF84KjP0tQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABXy38c/g7/wh8zeOvBDfYbGKZJLm2hfy2spS4CywkYwpcr8o5Qkbfl4T6kqO5toLy1ltbyGOe3mQxyxSoGSRSMFWB4IIOCDQB5t8FPipB8Q/C6Wup3cZ8SWKYvYtgj85d2FmUA4IIKhsYwxPCgrn02vkD4neB774J/EfTfE3hBLgaSZlntJJCxSGUE7rV2VtzKVB64LIzLlirGvpvwB41sfiB4NtNf09PJ87KT2xkV2t5VOGQkfgRkAlWU4GcUAdJRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABXyv+0pqs/in4paJ4P0hI7i4skWJYwCjm5uWXCFmIXG0QkEcDccnsPqivkTwZAnjH9rae6mMmr2cOrXd2txHI0iLHFvNu+9T9wMsIXnb91ehxQB9U+HdBsfC/huw0TSo9lpYwrDHlVDNjq7bQAWY5YnAyST3rSoooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAyPFXhnTvGPhe+0DWVkazvUCuYn2upDBlZT6hlBGcjjkEZFfKXgjX9f+AHxSm0XxNbxx6fevFHfkqWR4dxCXMThSzBdznAHPzKQGHy/YleU/Hn4Yp468Itqel28f9vaUjSxOsTNJcwqGLW428kknK8H5uBjexoA9StrmC8tYrqzmjnt5kEkUsThkkUjIZSOCCDkEVJXz3+zT8Tn1G1PgfXLiSS6tkaXTZppV+aFQAbcA4YleWX73y7h8oQZ+hKACio7m5gs7WW6vJo4LeFDJLLK4VI1AyWYngAAZJNYfhPx14b8c2txceFdUjv0tnCTAI8bxkjIyjgNg84OMHBAPBwAdBRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAV8k/sqf8AJU9S/wCwLL/6Pgr62r5J/ZU/5KnqX/YFl/8AR8FAH1tRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUVxvxE+KGgfDO1sZdfS8me+d1ghs4g7kIAWY7mVQBuUdc/MMA4OOb0r9pL4c6jatLeaheaU6uVEN5ZOzsMA7h5Qdcc45OeDx0yAeUfH3wJdeA/HFp498M/uLa8vVnZ2kEjQX+5pchGH3W27gPmGQ4OBtFfQHwy8d2vxE8D2uswfLcriC+iEZRYrhVUuFyTlfmBByeGGecgYd38R/hd8Q/Deq6Jc+KNPFpcQ+TN9sP2VhuztePz1ALKRuBAO0hT6V87/D3xh/wp34uXVr/aFvqWhzTfY7u5gn3xSQbwUuVETOCyqd235iAzpwSSAD3b9pHxT/AGB8KZdPgm2XetTLaKEuPLkEQ+eRgByykKI2HAxKM9cGj+zB4ZTSvhnNrbrGbjWrpmDo7E+TETGqsDwCHEp46hhk9h5b8dtTn8e/HW08MaZcR7LV4NLgZrkvD58jDe5AB2EM6owAJ/dc9MD6y03T7XSNKtNN0+LybSzhSCCPcW2IihVGSSTgAck5oAs0UUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFfJP7P3/ABSP7QF/4e1L97dvDd6YHt/mj82Jw7HJwduIGwcZyRwOcfW1fJOg/wDFM/tky/25/ovna1dbP48/aUk8j7ufvedH9N3OMHAB9bUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAc/wCLPAvhvxza29v4q0qO/S2cvCS7xvGSMHDoQ2DxkZwcAkcDHAar+zH8P9RullsxqmlIqBTDZ3YZGOSdxMqu2eccHHA4659fooA8A1f9k3Q5vJ/sHxLqFljd5v22BLnf0xt2+XtxznOc5HTHPP6r+yXqsNqraH4qs7y43gNHeWjW6BcHJDK0hJzjjA6nnjB+jR4j0RtcOjLrGnnVR1sRdJ5443f6vO7pz06c1pUAeJfB34BSeBNebxB4nvLe81SDcljHZO5ihDLtZ2LKpZiGZQMYAyeSRt9toooAKKxbjxdodr4utfC818BrN3CZ4rVYnYlACSxYAqv3T1I6VtUeYeQUVj+JvFejeDtLXUvEd4bOzaVYRL5LyAM2cA7FJA4PJ4rXVg6BkIKsMgjuKAFooooAKKKKACiiigAooooAKKKKACvkT43NP4O/aSt/Ec8cd0jPZapDAkhUssW1NjEj5SWgbkZ4IPtX13XzB+1ppUEOv+G9XVpDcXVrNaupI2BYmVlIGM5zM2eew6c5APp+isTwVqF1q/gHw/qWoS+dd3mmW088m0Lvd4lZjgAAZJPAGK26ACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACqWsXV7ZaNdXOk6f/aV7FGWhs/OWHzm7Lvbhfqau0HoaUtmNbnzd8BLi6nvvEOu3/g+2u5EvLm4k1yS4iMttLsyYFBBfncfmU4+avUNB+Kf9t/B298d/wBj+R9link+w/at27ys8eZsGM4/u8e9c58CvDurWHgjxTY6tp93ps15qU/lC8geIsrRqAwDAZGe4rjtBPjHRfgxrfw6/wCEE1qXVALlPtfk4tTC3LMsn8b8kKig7uMdxSbfK0v5Vb1Hb3r/AN539D0TV/jVHo/gTwr4mm0GWaPxBMsbW0NxueAHOSvyfvDxwPlz61Np3xavx4+07wz4r8HXXh/+11ZtOuJLyOYygZxvRR+7JA5GSQSB05rz7xH4Y16f4P8AwysoNE1KS6sr6JrqBLSQvbqCcl1Ayo+uK6/4oaLqmofGT4dXthpt5dWlndO1zPDAzxwDfHy7AYXoevpWtl7W3Tma+VkZXfs79eW/zuVPAG3XP2kvHurz/M+nRx2MAJJ2Lwpx6f6v9TXjdo3w5MXj1vHBlOvG9uTpZj8/duy+Mbf3f38ffr2T4ar/AGP+0P8AETSpvke88u+iXH31J3E/+RRXOeBNX1PwJL40sdZ+H/irVF1TUppIRaaS0kMqHcMMxx8pz1AIwe9Yf8u4/wCH8br8b3Nn8cv8S+635GzBZahr/wCyHdjxHfLqV0LGW5ScXCztiOQugLgkEjaAecjoeRXoPwl1aTW/hL4dvZ3Z5TZrE7N1JQlCf/Ha800Tw5qvw5/Zf8Ux+Jl+z3F5HPIlpvD/AGcSqsSqSDjJPJAPGfXIr0X4NadJpfwd8N28wIdrQTEMMEeYxcfowrol8U7f3fv1MUrKP/b33aHb0UUVmWFFFFABRRRQAUUUUAFFFFABXjf7UOn3V78I4p7aLfFY6nDPcNuA2IUkjB5PPzyIMDJ5z0Br2SsjxV4Z07xj4XvtA1lZGs71ArmJ9rqQwZWU+oZQRnI45BGRQBwH7Nuqwaj8FrG1gSRX026ntZi4ADMXM2VweRtlUc45B+p9Wrm/AngTR/h34b/sbQPtDQtM08ktzJvkldsDJwAB8qqMAAYX1yT0lABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUVRl1vSoNZh0ifU7OPU7hDJDZPcKJpF5+ZUJ3EfK3IH8J9KAL1FFFAHM3HgPS5/iNa+NUmuoNTgtjaukTqIp0IP+sBUkkZ4wR90eldNRRR0sHW5zvjfwXYePfDv9iavdXlvaNMksn2R1VpNvIUllb5c4PrwOa34IY7a3jggUJHEoRFHRQBgCn0UbBuFFFFABRRRQAUUUUAFFFFABXP+LPHfhrwNbwT+KdVjsFuGKwqUeRnIGThUBbA7nGBkeoroK+d/iQfGMn7SXhmKy/sNrhIpH0dbkSmIJhsmfHO7IONvHC+9LVyUe4/suXY+gNO1C11XTLbUNPlE1rdRLNDIARvRhkHB5GQe9WK88tfHWtH43R+CryCw+yjSFvJZYkff53AIBLY25zjK596j0r4h6tffEvxp4emt7MWmgWqzWrqj+Y7FA3zndgjJ7AU21083924lfr5L7z0esLxn4w0zwL4XuNd1vzjbQsq7IFDSSMxwAoJAJ79RwDXkem/Fv4la38MZvGOm6H4djs9OEhvDO8xa42tyYY1b5Aq9d7HJ5HpVf4reJYfHfhX4aBYfKt/EGqRTS2+7eVwQjLnjODIw6c+3Shxd0l3S+8E1v5P8D3Sz1SG60ODVZleyglt1uGF0VRoVK7vn5IBA684HrVXSvFvhzXbprbQ9f0vUrhUMjQ2d7HK4UEDOFJOMkc+9eR/GSaTxN8WvB3w+uZpYtGvMXV5FE+3zxubCnHoIz/31nqBXp9p8O/B+n6hZX2neG9NsruxcvBPa26wuDtK8lcFuGPDZHfrzTVpLm6Xa+4TvH3etk/vE8F+OtM8c2t/LpkF3ayafdtaXNveIqyI69eFZhj8exrpa8X+HrjS/2kviDpEA2wXUcV6VB4D/ACkn8TK1e0UlrGMu6G9JSj2YUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUVHc3MFnay3V5NHBbwoZJZZXCpGoGSzE8AADJJrx/wAX/tMeEPD809pocVx4gu4uA9uRHbFg5Vl805JwASGVWU5GDySAD2SvO/Gnxy8E+Cbo2d5fSalfI+yW00xVmeHBYHexYIpBXBUtuGR8uOa+f5vEfxi+Nclx/Y8d5/Zyo8UlvpzfZLPBVFeNnZgJCQQdjuxwzYABNegeD/2VtOto47jxxq0l7cB1Y2enNshwGOVaRhvcMNvQIR8wBPBoA4jxF8cfiJ8R7qXTfBVheadbqnmNbaMjz3RUFMs0qjcAGHVAnD7Turqfht+zz4hTxVZeK/HmpfZrm2vVvvsiSCeeaVXLZllyVGWCtwXLBiCVNfQGg+HdH8L6Umm+HtOt9PtEwfLgTG4hQu5j1ZiFGWYknHJNaVABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAV5h4p8Ia5qXx+8J+JLKx8zSdOtZI7q485B5bESYG0tuP3h0B616fRQtJKXYN4uPc8k8Y+GfGGkfGay8d+ENGh1+F7A2V1YtdpbOvXBDvxjlTxnoRjkGq3grwX4ztPH3jXW/FNjao2tWIWF7SdWjL7eIwCd3yjCksACQTXslFTyq1vVffuO+t/T8Njxnwd4C8S6V+zfrPha/03ytZuYrpYrXz423Fx8vzhioz7muR8ceHtS8H/Bz4dXmtWxt7nw7qam7hDq5QM5bqpKn7o6Z619KVna/oGmeKNEuNI121F3Y3AAlhLMu7BBHKkEcgdDVSbbuv7v/AJKTGKWj8/xOA+KfgHVvEuo6H4x8Ez2/9u6KfMhhuGxHdR/eC56ZzxyQCGOSOta2geIfiLq2uWkGseB7Tw/p65a7uptWjumcbcBY0j5DbiDlsjAPfFdlYWNvpmm21hZIUt7WJYYlZ2cqijAG5iSeB1JzVinpG6WwatXe54v8Mk/tr4/fELxDGCbeB009HByGZcK3/or9a9ori2m8B/BzQ5TLc2+iWt3O1wySzSTS3EhKqzKpLSPjK525wDk4HNeaeKf2q9HtPNg8IaLcajKPNRbq9byYgRwjqgyzqeSQfLOMDgnhLSMY9lb+vmN6ylLu/wCvwPf6xNe8aeGvC+8eIde0/T5UhM/kT3CrKyDPKx53NnaQAoJJGBk18t/8J38afivc7fDv9oQ2jTYUaPGbWCJ0jyVNwSCMg7tryYJYYH3RWv4f/ZS1+9tfN8R69Z6S7IjJDbwm6dSQSyucooK8D5SwPPPAyAfSPhnxVovjHRl1Xw1qEd/Zs7RmRAylWXqrKwDKehwQOCD0INa9c34E8CaP8O/Df9jaB9oaFpmnkluZN8krtgZOAAPlVRgADC+uSekoAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiio7m5gs7WW6vJo4LeFDJLLK4VI1AyWYngAAZJNAElFeZeJv2g/h/4bkaFNTk1i4R1VotKjEwAK7twkJWNgOAdrEgnGODjxHxB+0D8QPHGp/2Z4NtJNLS4R0Sz02I3N1KCg3fvNu7I2swMaoQD1OAaAPqDxP4y8PeDLBbvxPq1vp8T/6tZCWklwQDsjXLPjcudoOAcnArw3xp+1SkF0bbwFpMdykb4a+1NWCSAFgdkSkNg/KwZmB6goOtZGg/s2eL/E+qpqvxG1v7N52GuAbg3d6+1goUucoMoOG3Pj5Rt6ge3eB/hF4Q8AbZtG07z9QXP/ExvSJZ/wCIfKcAJ8rlTsC5GM560AfONn8P/ix8aZE1jW7iRbXZvtrnV3MEJyqf6mJFOAy7TuVArbTlia9l8H/s2eDfD0ccuurJ4ivldXElzmOFSrEjEKnBBBUMHLg7egBIr1+igCO2toLO1itbOGOC3hQRxRRIFSNQMBVA4AAGABUlFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFAHy3+1lpHk+KvD+s+fu+12Ulp5Oz7nkvv3bs858/GMcbe+ePMtI8IePNJkjvtM8IXl6l1apJFK+irqELxyKsisu6N0zgryPmHIOORX0t+0vpH9pfBu4uvP8r+y72C727M+bkmHbnPH+u3Z5+7jvkH7NGr/ANpfBu3tfI8r+y72e03b8+bkibdjHH+u245+7nvgAHkn/DTvxB0z/QNQ0zRzd2v7mc3VnKkpdeG3qJFAbIOQAADngdK6m2/a3ga6iW88GyRW5cCWSLUg7queSqmJQxA6AkZ9R1r6Nrlrn4Y+Bbq1lt5fB+hqkyFGMWnxRuARg7XUBlPoQQR1BoA4TT/2ofAV7fx29zDrGnxPndc3NqjRpgE8iN2bnGOFPJ7Dmup0r42fDnWbprez8V2cbqhcm8V7VMZA4eVVUnnoDnqccGsTVf2bfhzqNqsVnp95pTq4YzWd67OwwRtIlLrjnPAzwOeueW1f9k3Q5vJ/sHxLqFljd5v22BLnf0xt2+XtxznOc5HTHIB7bpHiPQ/EHnf2DrOn6n5G3zfsV0k3l7s43bScZwcZ9DWlXyvqv7J/iOG6VdD8Q6XeW+wFpLxJLdw2TkBVEgIxjnI6njjJo/8ACmfjT4R/0Dwxf3D2kn75zo+sm3i3ng5V2jJbCrzgjGOeMAA+tqK+Sft/7Q3g3/iX7PEFx5v7/f8AZk1TGflx5u2Tb937m4Y64+bJP+F5/F/wb/yNen7/ALX/AMe/9taS1vjb97Zs8vd95c5zjjpnkA+tqK+UtP8A2rvFUV/G+q6Fo9zaDPmRWwlhkbg4w7O4HOD905HHHUbf/DXP/Uk/+Vb/AO00AfSVFfNv/DXP/Uk/+Vb/AO01p2X7WWhvpVxJqHhrUINQXd5EEE6SxSfL8u6Q7SuWyDhGwOeelAHv9FfNv/DXP/Uk/wDlW/8AtNH/AA1z/wBST/5Vv/tNAH0lRXyT/wANV+N/+gV4f/8AAef/AOPVFLq37QfiySHS3j8SQFnLq6WQ04ZCn70wWMAYzwzYJxwTigD6y1DUrHSLCS+1W8t7G0ix5lxcyrHGmSAMsxAGSQPqa878TftB/D/w3I0KanJrFwjqrRaVGJgAV3bhISsbAcA7WJBOMcHHi1j+zn8R/FV+dR8WX9vZzSzKlxLqF4bq5ZAFG8bNwbC8AM6n5ccDBr0Twz+yz4X0yRZvEupXmuOrsREg+ywspXADBSXyDk5DjsMcHIBxvin9qvWLvzYPCGi2+nRHzUW6vW86Ug8I6oMKjDkkHzBnA5A5xP8AhBPjT8V7nd4i/tCG0abLHWJDawROkeAwtwARkHbuSPBLHJ+8a+pNB8F+GvC+w+HtB0/T5UhEHnwW6iVkGOGkxubO0EliSSMnJrboA8N8M/ss+F9MkWbxLqV5rjq7ERIPssLKVwAwUl8g5OQ47DHBz7BoPh3R/C+lJpvh7TrfT7RMHy4ExuIULuY9WYhRlmJJxyTWlRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQBkeLNKn17wXrekWbRpcahp89rE0pIRWeNlBYgE4yecA185fsm6v5PirxBo3kbvtdlHd+dv+55L7Nu3HOfPznPG3vnj6kqjY6LpWl3V3daZplnZ3F8/mXUtvbrG9w2SdzkAFjlmOTnqfWgC9RRRQAUUUUAFFFFABRRRQBW1DTbHV7CSx1Wzt760lx5lvcxLJG+CCMqwIOCAfqKxP8AhXHgj/oTfD//AIK4P/ia6SigDm/+FceCP+hN8P8A/grg/wDiazb34NfDy/1W31Cfwnp6zW+3YkCmGI7W3DdEhCPyedynI4ORxXbUUAc3/wAK48Ef9Cb4f/8ABXB/8TR/wrjwR/0Jvh//AMFcH/xNdJRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQB//Z)

From the above diagram fun(A) is calling for fun(B), fun(B) is calling for fun(C) and fun(C) is calling for fun(A) and thus it makes a cycle.

Text

Description automatically generated

Output : 20 19 9 8 4 3 1

Let’s understand the example by tracing tree of recursive function. That is how the calls are made and how the outputs are produced.

Diagram

Description automatically generated

70) Difference between Iterators vs Recursive methods?

Recursive: A program is called recursive when an entity calls itself.

Iterators: A program is called iterative when there is a loop.

|  |  |  |
| --- | --- | --- |
| **Property** | **Recursion** | **Iteration** |
| **Definition** | Function calls itself. | A set of instructions repeatedly executed. |
| **Application** | For functions. | For loops. |
| **Termination** | Through base case, where there will be no function call. | When the termination condition for the iterator ceases to be satisfied. |
| **Usage** | Used when code size needs to be small, and time complexity is not an issue. | Used when time complexity needs to be balanced against an expanded code size. |
| **Code Size** | Smaller code size | Larger Code Size. |
| **Time Complexity** | Very high(generally exponential) time complexity. | Relatively lower time complexity(generally polynomial-logarithmic). |

71) std::allocator and deallocators in c++?(https://www.geeksforgeeks.org/stdallocator-in-cpp-with-examples/)

Allocators are objects responsible for encapsulating memory management. Std::allocator is used when you want to separate allocation and do construction in two steps. It is also used when separate destruction and deallocation are done in two steps. All the STL containers in c++ have a type parameter allocator that is default std::allocator. The default allocator simply uses the operator’s new and delete to obtain and release memory.

Text

Description automatically generated

**Advantage of using std::allocator**

* allocator is the memory allocator for the STL containers. This container can separate the memory allocation and de-allocation from the initialization and destruction of their elements. Therefore, a call of vec.reserve(n) of a vector vec allocates only memory for at least n elements. The constructor for each element will not be executed.
* allocator can be adjusted according to the container of your need, for example, vector where you only want to allocate occasionally.
* On the contrary, [new](https://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/) doesn’t allow to have control over which constructors are called and simply constructs all objects at the same time. That’s an advantage of std:: allocator over new

72) what is the placement new? (https://www.geeksforgeeks.org/placement-new-operator-cpp/)

Placement new is a variation new operator in c++. Normal new operator does two things

i) Allocates memory ii) constructs an object in allocated memory.

Placement new allows us to separate the above two things. In placement new, we can pass a preallocated memory and construct an object in the passed memory.

**new vs placement new**

* Normal new allocates memory in the heap and constructs objects there whereas using placement new, object construction can be done at a known address.
* With normal new, it is not known that, at what address or memory location it’s pointing to, whereas the address or memory location that it’s pointing to is known while using placement new.
* The deallocation is done using [delete](https://www.geeksforgeeks.org/g-fact-30/) operation when allocation is done by new but there is no placement delete, but if it is needed one can write it with the help of [destructor](https://www.geeksforgeeks.org/playing-with-destructors-in-c/)

**When to prefer using placement new?**

As it allows to the construction of an object on memory that is already allocated, it is required for optimizations as it is faster not to re-allocate all the time. There may be cases when it is required to reconstruct an object multiple time. The placement new operator might be more efficient in these cases.

Example:

Text

Description automatically generated

**How to delete the memory allocated by placement new?**

The operator delete can only delete the storage created in the heap, so when placement new is used delete operator cannot be used to delete the storage. In the case of memory allocation using the placement new operator, since it is created in the stack the compiler knows when to delete it and it will handle the deallocation of the memory automatically. If required, one can write it with the help of a [destructor](https://www.geeksforgeeks.org/playing-with-destructors-in-c/) as shown below.

**When will placement new operator show segmentation fault?**

The placement new operator should be used with care. The address which is passed can be a reference or a pointer pointing to a valid memory location. It may show an error when the address passed is : 

* A pointer such as NULL pointer.
* A pointer that is not pointing to any location.
* It cannot be a void pointer unless it points to some location.

Text

Description automatically generated

**Advantages of placement new operator over new operator**

* The address of memory allocation is known before hand.
* Useful when building a memory pool, a garbage collector or simply when performance and exception safety are paramount.
* There’s no danger of allocation failure since the memory has already been allocated, and constructing an object on a pre-allocated buffer takes less time.
* This feature becomes useful while working in an environment with limited resources.

73) What is ODR (One Definition Rule)?

* Within a given file, a function, variable, type, or template can only have one definition.
* Within a given program, a variable or normal function can only have one definition. This distinction is made because programs can have more than one file (we’ll cover this in the next lesson).
* Types, templates, inline functions, and inline variables are allowed to have identical definitions in different files. We haven’t covered what most of these things are yet, so don’t worry about this for now -- we’ll bring it back up when it’s relevant.

**Violating part 1 of the ODR will cause the compiler to issue a redefinition error. Violating ODR part 2 will likely cause the linker to issue a redefinition error. Violating ODR part 3 will cause undefined behavior.**

**74) What is Naming collisions and when it will happen?**

* + Two or more identically named functions are introduced into separate files belonging to the same program. This will result in a linker error
  + Two or more identically named functions are introduced into the same file. This will result in a compiler error.

75) Difference between a syntax error and a semantic error?

A **syntax error** occurs when you write a statement that is not valid according to the grammar of the C++ language. This includes errors such as missing semicolons, using undeclared variables, mismatched parentheses or braces, etc…

Example: std::cout < "Hi there"; << x; // invalid operator (<), extraneous semicolon, undeclared variable (x)

A **semantic error** occurs when a statement is syntactically valid but does not do what the programmer intended.

**Example:** int x;

std::cout << x; // Use of uninitialized variable leads to undefined result

**76) what are the storage classes?**

|  |  |  |
| --- | --- | --- |
| **Specifier** | **Meaning** | **Note** |
| extern | static (or thread\_local) storage duration and external linkage |  |
| static | static (or thread\_local) storage duration and internal linkage |  |
| thread\_local | thread storage duration |  |
| mutable | object allowed to be modified even if containing class is const |  |
| auto | automatic storage duration | Deprecated in C++11 |
| register | automatic storage duration and hint to the compiler to place in a register | Deprecated in C++17 |

77) Explain pointer const rules and examples?

* A non-const pointer can be assigned another address to change what it is pointing at
* A const pointer always points to the same address, and this address can not be changed.
* A pointer to a non-const value can change the value it is pointing to. These can not point to a const value.
* A pointer to a const value treats the value as const when accessed through the pointer, and thus can not change the value it is pointing to. These can be pointed to const or non-const l-values (but not r-values, which don’t have an address)

Keeping the declaration syntax straight can be a bit challenging:

* The pointer’s type defines the type of the object being pointed at. So a const in the type means the pointer is pointing at a const value.
* A const after the asterisk means the pointer itself is const and it can not be assigned a new address.

Example:

int main()

{

const int x { 5 }; // x is now const

int\* ptr { &x }; // compile error: cannot convert from const int\* to int\*

const int\* ptr { &x }; // okay: ptr is pointing to a "const int"

\*ptr = 6; // not allowed: we can't change a const value

const int y{ 6 };

ptr = &y; // okay: ptr now points at const int y\*/

int x1 { 6 };

int y1 { 10 };

int\* const ptr1 { &x1 }; // okay: the const pointer is initialized to the address of x

ptr1 = &y1; // error: once initialized, a const pointer can not be changed.

return 0;

}

**Example2:**

int main()

{

int value { 5 };

int\* ptr0 { &value }; // ptr0 points to an "int" and is not const itself, so this is a normal pointer.

const int\* ptr1 { &value }; // ptr1 points to a "const int", but is not const itself, so this is a pointer to a const value.

int\* const ptr2 { &value }; // ptr2 points to an "int", but is const itself, so this is a const pointer (to a non-const value).

const int\* const ptr3 { &value }; // ptr3 points to an "const int", and it is const itself, so this is a const pointer to a const value.

return 0;

}

78) Pass by address… by reference?

Just like we can pass a normal variable by reference, we can also pass pointers by reference. Here’s the same program as above with ptr2 changed to be a reference to an address:

#include <iostream>

void nullify(int\*& refptr) // refptr is now a reference to a pointer

{

refptr = nullptr; // Make the function parameter a null pointer

}

int main()

{

int x{ 5 };

int\* ptr{ &x }; // ptr points to x

std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");

nullify(ptr);

std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n");

return 0;

}

**Because references to pointers are fairly uncommon, it can be easy to mix up the syntax (is it int\*& or int&\*?).**

**The good news is that if you do it backward, the compiler will error because you can’t have a pointer to a reference (because pointers must hold the address of an object, and references aren’t objects). Then you can switch it around**.

78) Explain about Memory layout?

The memory that a program uses is typically divided into a few different areas, called segments:

* **The code segment (also called a text segment)**, where the compiled program sits in memory. The code segment is typically read-only.
* **The bss segment (also called the uninitialized data segment),** where zero-initialized global and static variables are stored.
* **The data segment (also called the initialized data segment),** where initialized global and static variables are stored.
* **The heap,** where dynamically allocated variables are allocated from.
* **The call stack,** where function parameters, local variables, and other function-related information are stored.

79) stack advantages and disadvantages?

The stack has advantages and disadvantages:

* Allocating memory on the stack is comparatively fast.
* Memory allocated on the stack stays in scope as long as it is on the stack. It is destroyed when it is popped off the stack.
* All memory allocated on the stack is known at compile time. Consequently, this memory can be accessed directly through a variable.
* Because the stack is relatively small, it is generally not a good idea to do anything that eats up lots of stack space. This includes passing by value or creating local variables of large arrays or other memory-intensive structures.

80) What is OOP?

Object Object-oriented provides us with the ability to create objects that tie together both properties and behaviors into a self-contained, reusable package.

81) What is Encapsulation?

Encapsulation is the process of keeping the details about how the object is implemented hidden away from users of the object. Instead, users of the object access the object through a public interface.

Advantages:

* Encapsulated classes are easier to use and reduce the complexity of your programs.
* Encapsulated classes help protect your data and prevent misuse.
* Encapsulated classes are easier to change.
* Encapsulated classes are easier to debug.

82) what are the Delegating constructors?

Constructors are allowed to call other constructors from the same class. This process is called delegating constructors (or constructor chaining).

Example:

class Foo

{

private:

public:

Foo()

{

// code to do A

}

Foo(int value): Foo{} // use Foo() default constructor to do A

{

// code to do B

}

};

83) What is the Emplace?

* All the containers support inserts and emplace operations to store data.
* Emplace is used to construct object in-place and avoids unnecessary copy of objects.
* Insert and Emplace is equal for primitive data types but when we deal with heavy objects we should use emplace if we can for efficiency.

76) what is hyper threading? (https://www.guru99.com/cpu-core-multicore-thread.html)

https://www.geeksforgeeks.org/commonly-asked-operating-systems-interview-questions/

<https://www.geeksforgeeks.org/last-minute-notes-operating-systems/>

https://www.simplilearn.com/tutorials/git-tutorial/git-interview-questions

https://www.interviewbit.com/git-interview-questions/

https://aticleworld.com/dynamic-memory-and-new-operator-c/

/////////////\*\*\*\*\*\*\* Exception Handling \*\*\*\*\*\*\*\*\*\*\*\*///////////////////////

1) Exceptions: Exception handling provides a mechanism to decouple the handling of errors or other exceptional circumstances from the typical control flow of your code. This allows more freedom to handle errors when and however is most useful for a given situation, alleviating most of the messiness that returns code cause.

2) Performance concerns

Exceptions do come with a small performance price to pay. They increase the size of your executable, and they may also cause it to run slower due to the additional checking that has to be performed. However, the main performance penalty for exceptions happens when an exception is actually thrown. In this case, the stack must be unwound and an appropriate exception handler found, which is a relatively expensive operation.

As a note, some modern computer architectures support an exception model called zero-cost exceptions. Zero-cost exceptions, if supported, have no additional runtime cost in the non-error case (which is the case we most care about performance). However, they incur an even larger penalty in the case where an exception is found.

1. What is difference between double and long long double?