**PARAMETER PASSING**

**Pass by value** - The function receives a copy of the actual parameter's value. Changes to the formal parameter do not affect the actual parameter.

**Pass by reference** - The function receives a reference to the actual parameter. Changes to the formal parameter reflect on the actual parameter. Uses the & symbol.

void function(int &x) {

x = 10;

}

int main() {

int a = 5;

function(a);

cout << a; // Outputs 10

}

**Pass by pointer** - The function receives a pointer to the actual parameter. Changes via the pointer affect the actual parameter.

void function(int \*x) {

\*x = 10;

}

int main() {

int a = 5;

function(&a);

cout << a; // Outputs 10

}

**Const parameter** - Ensure that a parameter passed by reference or pointer isn't unintentionally modified.

void safeFunction(const int &x) {

// x cannot be modified

}

**DYNAMIC MEMORY**

Dynamic memory, also known as heap memory or free store, allows for memory allocation during runtime. This contrasts with stack memory, which is determined at compile-time.

**Operations:**

1. Allocation: Requesting a block of memory during runtime.
2. Deallocation: Releasing the memory back to the system when it's no longer needed.
3. Access: Reading or writing data to and from dynamically allocated memory.

**Allocation:**

new: Allocates memory for a single object. Returns a pointer to the object's type.

int\* ptr = new int(5); // Allocates an integer initialized with 5

new[]: Allocates memory for an array.

int\* arr = new int[10]; // Allocates an array of 10 integers

**Deallocation:**

delete: Deallocates memory allocated by new.

delete ptr;

delete[]: Deallocates memory allocated by new[].

delete[] arr;

**Access:**

Like any other pointer, use the \* dereference operator for single objects and array indexing for arrays.

\*ptr = 10;

arr[2] = 20;

**Memory Leaks:**

**Definition**: Memory that's been allocated dynamically but hasn't been deallocated and is no longer accessible is termed a "memory leak". Over time, these leaks can accumulate, leading to excessive memory usage and potential program crashes.

Prevention Strategies:

* Diligence: Always pair every new with a corresponding delete and new[] with delete[].
* Scope: Use local scope, RAII (Resource Acquisition Is Initialization), and objects to manage memory. When objects go out of scope, their destructors can deallocate memory.

**Pointers as Arrays**

In C++, an array name is essentially a constant pointer to the first element of the array. Because of this relationship, pointer arithmetic and array indexing can be used interchangeably.

**Accessing Elements**

If you have a pointer pointing to the start of a block of memory (like an array), you can use array-like indexing on the pointer to access the elements.

int arr[5] = {1, 2, 3, 4, 5};

int\* p = arr;

cout << p[2]; // This will output 3

Similarly, array names can be dereferenced using pointer notation.

cout << \*(arr + 2); // This will also output 3

**Dynamic Arrays with Pointers**

You can dynamically allocate an array using pointers.

int\* dynamicArray = new int[5]; // Dynamically allocates an array of 5 integers

You can then use this pointer as if it were an array:

for (int i = 0; i < 5; i++) {

dynamicArray[i] = i \* i; // Assigns square of i to each element

}

**Pointer Arithmetic**

You can perform arithmetic on pointers. This is especially useful when working with arrays.

int arr[5] = {10, 20, 30, 40, 50};

int\* p = arr;

p++; // Now p points to the second element, i.e., 20

cout << \*p; // Outputs 20

**Multidimensional Arrays and Pointers**

For multi-dimensional arrays, things get a bit more complex. A 2D array, for instance, can be thought of as an array of arrays. You can use a pointer to a pointer to represent a 2D array.

int\*\* matrix = new int\*[rows];

for (int i = 0; i < rows; i++) {

matrix[i] = new int[cols];

}

This dynamically allocates a 2D array. Note that you'd have to loop through each row to deallocate each sub-array, and then finally deallocate the array holding the pointers.

**DYNAMIC 1D AND 2D ARRAYS**

Dynamic arrays are arrays whose size can be determined during runtime, as opposed to compile-time.

**Size Variable:** Always keep an auxiliary variable to track the size of the dynamic array since pointers don't store this information.

**Declare:**

int\* dynamic1DArray;

int\*\* dynamic2DArray;

**Define/allocate:**

dynamic1DArray = new int[10]; // 1D array of 10 integers

dynamic2DArray = new int\*[5]; // 2D array: an array of 5 integer pointers

for (int i = 0; i < 5; i++) {

dynamic2DArray[i] = new int[5]; // Each row has 5 columns

}

**Deallocate:**

For 1D:

delete[] dynamic1DArray;  
2D:

for (int i = 0; i < 5; i++) {

delete[] dynamic2DArray[i];

}

delete[] dynamic2DArray;

**POINTERS**

1. Declaring Pointers

To declare a pointer, you use the asterisk (\*) operator. The type of the pointer corresponds to the type of variable it's pointing to.

int \*ptr; //Here, ptr is a pointer to an integer.

2. Initializing Pointers

You can initialize a pointer using the address of a variable. The & operator is used to get the address of a variable.

int x = 10;

int \*ptr = &x; //Here, ptr now holds the address of x.

3. Accessing Value Using Pointers

The \* operator is a dereference operator, and it's used to access the value at the address held by the pointer.

int x = 10;

int \*ptr = &x;

std::cout << \*ptr; // Outputs: 10

4. Pointer Arithmetic

You can perform arithmetic operations on pointers. This is particularly useful when dealing with arrays.

int arr[5] = {1, 2, 3, 4, 5};

int \*ptr = arr; // Pointing to the first element

ptr++; // Move to the next element

std::cout << \*ptr; // Outputs: 2

5. Pointer to Pointer

A pointer can also store the address of another pointer.

int x = 10;

int \*ptr1 = &x;

int \*\*ptr2 = &ptr1; // Pointer to an integer pointer

6. Null Pointer

It's good practice to initialize pointers if they aren't immediately assigned a value. A null pointer does not point to any memory location.

int \*ptr = nullptr; // Using C++11 and later

**TYPE CASTING**

***1. Implicit Conversion (Automatic Type Conversion)***

This is when the compiler automatically converts one data type into another without the programmer's intervention. This usually happens in mixed-type expressions.

int i = 10;

double d = i; // i is automatically converted to double

Rules:

* Small data types can be converted to larger data types (e.g., int to double).
* Non-const can be converted to const.

If there's a risk of data loss, or when converting from larger to smaller types, C++ may give a warning.

***2. Explicit Conversion (Type Casting)***

C-style Cast:

double d = 10.5;

int i = (int)d; // d is explicitly converted to int

C++ Casts:

C++ introduces four named cast operators which offer better clarity and control compared to C-style casts.

static\_cast: Used for most common type-to-type conversions.

double d = 10.5;

int i = static\_cast<int>(d);

**ASCII**

ASCII is a character encoding standard that represents each character as a number. In the ASCII standard, every character is assigned a unique 7-bit integer between 0 and 127. For instance, the ASCII value of 'A' is 65, and the ASCII value of 'a' is 97.

Get ASCII value of a character:

char ch = 'A';

int asciiValue = static\_cast<int>(ch);

std::cout << "ASCII value of " << ch << ": " << asciiValue << std::endl;

// Outputs: ASCII value of A: 65

Convert ASCII value to a character:

int asciiVal = 97;

char character = static\_cast<char>(asciiVal);

std::cout << "Character for ASCII " << asciiVal << ": " << character << std::endl;

// Outputs: Character for ASCII 97: a

Converting lowercase to uppercase (and vice versa) by leveraging their ASCII difference:

char lower = 'c';

char upper = lower - ('a' - 'A');

std::cout << "Uppercase of " << lower << " is " << upper << std::endl;

// Outputs: Uppercase of c is C

**STD::STRING (#include <string>)**

Accessing Characters:

char c = s2[1]; // 'e' (using array-like indexing)

char d = s2.at(1); // 'e' (using the `at` method, which throws out\_of\_range exception on invalid access)

Modifying Characters:

s2[1] = 'E'; // s2 becomes "HEllo"

s2.at(1) = 'e'; // s2 becomes "Hello"

String Size and Capacity

size\_t length = s2.size(); // or s2.length(); Returns the number of characters

bool isEmpty = s2.empty(); // Returns true if the string is empty

Appending and Insertion

s2 += " World"; // Appending using += operator

s2.append("!"); // Appending using `append` method

s2.insert(5, " dear"); // Inserting "dear" before the 6th character

Finding and Replacing

size\_t pos = s2.find("World"); // Returns starting position of the first match or `std::string::npos` if not found

s2.replace(pos, 5, "Universe"); // Replaces "World" with "Universe"

Substrings

std::string subs = s2.substr(0, 5);

// Returns a substring starting from position 0 with length 5

C-String Conversion

const char\* cstr = s2.c\_str(); // Returns a null-terminated C-string

**STD::VECTOR (#include <vector>)**

Initialization

std::vector<int> v1; // Default initialization

std::vector<int> v2(5, 100); // Initialize with 5 integers all of value 100

std::vector<int> v3 = {1, 2, 3, 4, 5}; // Initialization with initializer list

std::vector<int> v4(v3.begin(), v3.end()); // Initialize with a range

std::vector<int> v5(v3); // Copy initialization

Capacity and Size Management

size\_t size = v1.size(); // Returns the number of elements

size\_t capacity = v1.capacity(); // Returns the total capacity

bool isEmpty = v1.empty(); // Returns true if the vector is empty

v1.resize(10); // Resize to contain 10 elements

v1.reserve(100); // Reserve space for at least 100 elements

v1.shrink\_to\_fit(); // Reduces capacity to fit the size

Modifying Elements

v1.push\_back(42); // Appends an integer with the value 42

v1.emplace\_back(42); // Constructs an element in-place at the end

v1.pop\_back(); // Removes the last element

v1.insert(v1.begin(), 0); // Inserts 0 at the beginning

v1.erase(v1.begin()); // Erases the first element

v1.clear(); // Removes all elements

Accessing Elements

int first = v2[0]; // Using array-like indexing (no bounds checking)

int second = v2.at(1); // Using the `at` method (throws out\_of\_range exception on invalid access)

int last = v2.back(); // Access the last element

int firstAgain = v2.front(); // Access the first element

Other Useful Operations

bool areEqual = (v2 == v3); // Compares if two vectors have the same elements in the same order

v2.swap(v3); // Swaps the contents of v2 and v3

for(const auto& elem : v2) {

// Use elem

}

**Homework Code**

/\*\*

\* TODO: [suggested] Student implement this function

\* Allocate the 2D map array.

\* Initialize each cell to TILE\_OPEN.

\* @param maxRow Number of rows in the dungeon table (aka height).

\* @param maxCol Number of columns in the dungeon table (aka width).

\* @return 2D map array for the dungeon level, holds char type.

\*/

char\*\* createMap(int maxRow, int maxCol) {

if (maxRow <= 0 || maxCol <= 0) {

return nullptr;

}

char \*\*map = new char\*[maxRow];

for (int i = 0; i < maxRow; i++) {

map[i] = new char[maxCol];

}

for (int i = 0; i < maxRow; i++) {

for (int j = 0; j < maxCol; j++) {

map[i][j] = TILE\_OPEN;

}

}

return map;

}

/\*\*

\* TODO: Student implement this function

\* Deallocates the 2D map array.

\* @param map Dungeon map.

\* @param maxRow Number of rows in the dungeon table (aka height).

\* @return None

\* @update map, maxRow

\*/

void deleteMap(char\*\*& map, int& maxRow) {

int temp = maxRow;

maxRow = 0;

if (map == nullptr) {

return;

}

for (int i = 0; i < temp; i++) {

delete[] map[i];

}

delete[] map;

map = nullptr;

}

/\*\*

\* TODO: Student implement this function

\* Resize the 2D map by doubling both dimensions.

\* Copy the current map contents to the right, diagonal down, and below.

\* Do not duplicate the player, and remember to avoid memory leaks!

\* You can use the STATUS constants defined in logic.h to help!

\* @param map Dungeon map.

\* @param maxRow Number of rows in the dungeon table (aka height), to be doubled.

\* @param maxCol Number of columns in the dungeon table (aka width), to be doubled.

\* @return pointer to a dynamically-allocated 2D array (map) that has twice as many columns and rows in size.

\* @update maxRow, maxCol

\*/

char\*\* resizeMap(char\*\* map, int& maxRow, int& maxCol) {

if (maxRow <= 0 || maxCol <= 0 || maxRow > 999999 || maxCol > 999999 || map == nullptr) {

return nullptr;

}

int newMaxRow = 2 \* maxRow;

int newMaxCol = 2 \* maxCol;

char\*\* newMap = new char\*[newMaxRow];

for (int i = 0; i < newMaxRow; i++) {

newMap[i] = new char[newMaxCol];

}

for (int i = 0; i < maxRow; i++) {

for (int j = 0; j < maxCol; j++) {

newMap[i][j] = map[i][j];

}

}

for (int i = 0; i < maxRow; i++) {

for (int j = maxCol; j < newMaxCol; j++) {

if (map[i][j - maxCol] == TILE\_PLAYER) {

newMap[i][j] = TILE\_OPEN;

continue;

}

newMap[i][j] = map[i][j - maxCol];

}

}

for (int i = maxRow; i < newMaxRow; i++) {

for (int j = 0; j < maxCol; j++) {

if (map[i - maxRow][j] == TILE\_PLAYER) {

newMap[i][j] = TILE\_OPEN;

continue;

}

newMap[i][j] = map[i - maxRow][j];

}

}

for (int i = maxRow; i < newMaxRow; i++) {

for (int j = maxCol; j < newMaxCol; j++) {

if (map[i - maxRow][j - maxCol] == TILE\_PLAYER) {

newMap[i][j] = TILE\_OPEN;

continue;

}

newMap[i][j] = map[i - maxRow][j - maxCol];

}

}

for (int i = 0; i < maxRow; i++) {

delete[] map[i];

}

delete[] map;

maxRow \*= 2;

maxCol \*= 2;

return newMap;

}

/\*\*

\* TODO: Student implement this function

\* Load representation of the dungeon level from file into the 2D map.

\* Calls createMap to allocate the 2D array.

\* @param fileName File name of dungeon level.

\* @param maxRow Number of rows in the dungeon table (aka height).

\* @param maxCol Number of columns in the dungeon table (aka width).

\* @param player Player object by reference to set starting position.

\* @return pointer to 2D dynamic array representation of dungeon map with player's location., or nullptr if loading fails for any reason

\* @updates maxRow, maxCol, player

\*/

char\*\* loadLevel(const string& fileName, int& maxRow, int& maxCol, Player& player) {

ifstream file(fileName);

if (!file.is\_open()) {

return nullptr;

}

long long tempRow, tempCol;

file >> tempRow >> tempCol;

if(file.fail()) return nullptr;

if (tempRow > 2147483647 / tempCol || tempCol > 2147483647 / tempRow) {

return nullptr;

}

if (tempRow \* tempCol > 2147483647) {

return nullptr;

}

maxRow = static\_cast<int>(tempRow);

maxCol = static\_cast<int>(tempCol);

file >> tempRow >> tempCol;

if(file.fail()) return nullptr;

if (tempCol != 0) {

if (tempRow > 2147483647 / tempCol) {

return nullptr;

}

}

if (tempRow != 0) {

if (tempCol > 2147483647 / tempRow) {

return nullptr;

}

}

player.row = static\_cast<int>(tempRow);

player.col = static\_cast<int>(tempCol);

if (maxRow <= 0 || maxCol <= 0 || maxRow > 999999 || maxCol > 999999 || player.row < 0 || player.col < 0 || player.row >= maxRow || player.col >= maxCol) {

return nullptr;

}

char\*\* map = createMap(maxRow, maxCol);

auto isValidTile = [](char tile) {

return tile == TILE\_OPEN || tile == TILE\_TREASURE || tile == TILE\_AMULET ||

tile == TILE\_MONSTER || tile == TILE\_PILLAR || tile == TILE\_DOOR ||

tile == TILE\_EXIT;

};

bool foundDoor = false;

for (int i = 0; i < maxRow; i++) {

for (int j = 0; j < maxCol; j++) {

if (file >> map[i][j]) {

if (file.eof() || file.fail() || file.bad()) {

deleteMap(map, maxRow);

return nullptr;

}

if (map[i][j] == TILE\_DOOR || map[i][j] == TILE\_EXIT) {

foundDoor = true;

}

if (!isValidTile(map[i][j])) {

deleteMap(map, maxRow);

return nullptr;

}

}

}

}

if (!foundDoor) {

deleteMap(map, maxRow);

return nullptr;

}

string remainingData;

getline(file, remainingData);

while (getline(file, remainingData)) {

for (char ch : remainingData) {

if (!isspace(ch)) {

deleteMap(map, maxRow);

return nullptr;

}

}

}

map[player.row][player.col] = TILE\_PLAYER;

file.close();

return map;

}