





1 -Syntax explanation

int\* ptr = &var;

-it shows that we declare a pointer that points to the address of the variable(var);

It stores it.

-Декларираме и инициализираме пойнтер-a ptr, който пази адреса на променливата var

2-Concept Clarification

difference between a while loop and do-while

-The difference is that a while loop checks the condition before it executes the code inside the while. And the do-while loop checks the condition after executing the code inside the do

-Разликата между двете е, че при обикновения while, преди изпълнението на кода в него, се проверява условието, а при do-while първо се изпълнява кода в do, и после се проверява условието

3-Code Analysis:

for(int i = 1; i <= 10; i++){

cout << i << endl;

}

so in here we use the for loop to iterate through the numbers from 1 to 10.

and for all of them we print the current number

we go through the numbers exactly 9 times, and every time we increment to get the next with 1, we end the loop when we reach the end condition

we end the loop when we reach the number 10

тук използваме for-а за да можем да обходим числата от едно до 10, като всеки път когато се изпълнява за всяко число, принтираме самото него и отиваме към следващото като му добавим 1. For-a се изпълнява точно 9 пъти,докато не стигне условието, че числото е стигнало до нашата цел.

4-Function Usage

We use functions so we can make the code more practical of use. Like this we don't need to write the same 5 lines of code like 10 times. This way it is even easier to read and understand. We can unify more code like this so everytime we can pass it a different variable, we can even make the function to get different types of variables

Използваме функциите, за да можем да направим кода си по-практичен и да може да го ползвам, за различни случай, където например всеки път ще разглежда различен масив, от числа но условията,за които проверява функцията да са едни и същи. Така не трябва да пишем един и същи код от 5 реда 10 пъти.

int square(int number){

return number \* number;

}

5-Memory Management

The difference between stack and heap memory allocation is that in the stack the memory is stored is destroyed automatically. And with the heap the programmer has to always destroy it so there is no data leak and has to be careful with allocating the memory. With the stack there is no such problem as the compiler is taking care of it on it's own. The stack destroys the info at the end of every execution of method/function(it empties itself).

We use the word "new <name of variable>" when we want to crete an object in the heap, and "delete <name of variable>" to delete. we cannot use the allocated heap memory after it is deleted :

int\* ptr = new int[3];

delete[] p;

Разликата между двете е, че при първата (stack-програмния стек), заделената памет(която се заделя в момента на дефиницията) за изпълнението на кода се освобождава автоматично при изход от частта(блока/функцията), в която се намира променливата, а при динамичната(heap-a), трябва да се изтрие ръчно, понеже иначе може да има изтичане на данни. Тя може да се заделя и освобождава по всяко време на изпълнение на програмата. Използваме думата "new" а създадем обект в динамичната памет, и delete а да я освободим. Не можем да достъпим памет след нейното освобождаване.

6-Encapsulation

So we use encapsulation to pack the data members nd functions of an object, so the user has no direct access to them, we usually do it by using getters, setters and we can change the data by methods, but no directly only through them. By using access modifiers the encapsulation is achieved using classes.

Енкапсулацията е един от основните принципи на ооп. Чрез нея капсулираме данните и методите на даден обект във клас.Потребителя няма директен достъп до член данните, но може да получи информация за тях и да ги променя, чрез гетъри, сетъри и ъпдейт функции. Използваме спецификатори за достъп, за да можем да отделим, какво може, и какво не може да достъпва потребителя(private-потребителя няма достъп, но член функциите-да, protected-ако има наследяване дъщерния клас може да достъпва член данните, но потребителя-не, public-обикновенно методите са от такъв тип, чрез тях потребителя взаимодейства с класа)

#include <iostream>

using namespace std;

class Point

{

public:

Point();

Point (int x, int y): x(x), y(y){}

int getX() const{

return this->x;

}

void setX(int x){

this->x = x;

}

int getY() const{

return this->y;

}

void setY(int y){

this->y = y;

}

void print()const {

cout << "X:" << getX() << endl;

cout << "Y:" << getY() << endl;

}

private:

int x;

int y;

};

int main() {

Point p1(0, 0);

Point p2;

p2.setX(1);

p2.setY(2);

p1.print();

p2.print();

}

7-Error Handling

Exceptions are used when the code isn't compiling as it should and it throws an exception, it may be due to not working properly with the code, or a mistake with the wrong input.

Ексепшъни се използват когато обикновено кода може да не се компилира поради грешка, която може да направи потребителя, или недобре написан код. Използва се try-catch метода, където в try слагаме кода, който ще се тества, ако в него кодът се компилира, няма да се throw-нат грешки, ако има проблем с изпълнението ще се.

В catch слагаме кода, който ще се изпълни ако има някаква грешка в try .

#include <exception>//needed library for exceptions

try{

string filename;

ifstream fin(filename);

if (!fin) throw runtime\_error("File not open");

else{

//do something

fin.close();

}

}catch(...){ //...it handles any type of mistake

cerr << "No such file in directory! Create a new one.";

}

8-STL Containers

So we can create type of structure that holds elements that can be of any type, like objects, pointers to objects, even int, string. The vector is a very useful stl structure, because it has lots of implemented functions, so we don’t need to create a new class just to hold the objects. We can use them directly.

Обект от Std::vector е структура, която може да съдържа всякакъв тип променливи, обекти, пойнтъри към обекти. Вектора е много полезна структура от данни понеже не трябва да създаваме клас, който да ни държи обектите,сам си алокира паметта, и размера, добавя нов, изтрива, сортира, дефакто има много вградени функции, които може да се използват наготово вместо да пишем нови.

#include <iostream>

using namespace std;

int main()

{

vector<int> v;

v.push\_back(3);

v.push\_back(5);

for(int i = 0; i < v.size(); i++){

cout << v.at(i) += v.at(i) << endl;

}

return 0;

}

9- Pointers and references

Pointers are used so we can store the address of a variable, and references we can use to minimise the memory used in our program when we pass a variable through a reference in a function so we don't set aside more memory for the program to run. When we use reference we create a new name for existing variable and we cannot change it to another variable. we can also pass by reference by pointers in a function but we need to dereference it to make changes to the copy.

Пойнтерите ги използваме за да можем да записваме адресите на други променливи, и да заделяме динамична памет. Когато подаваме променлива по референция във функция, ние заделяме по-малко памет отколкото да си подадем самия обект във функцията(така се създава копие на обекта). Използваме референциите също да зададем ново име, към съществуващ обект, но не можем да променим да сочи друг обект. Във функциите можем да подаваме и по референция, чрез поинтъри, но неудобството име е, че за да можем да променим нещо по тях, трябва да редеференсираме самата променлива във функцията.

#include <iostream>

using namespace std;

void sum(int a, int b, int& result) {

result = a + b;

}

void sum2(int a, int b, int\* result) {

\*result = a + b;

}

int main() {

int num1 = 5;

int num2 = 7;

int result;

sum(num1, num2, result);

cout << "Sum of " << num1 << " and " << num2 << " is: " << result << endl;

int result2;

sum2(num1, num2, &result2);

return 0;

}

10-Advanced Features:

Templates

We use it so we can pass different data type at functions, or even classes, so we don't need to make new ones for different data types.

Използваме ги, за да можем да генерализираме функциите, дори и класовете. Така например не трябва да правим принт функция за два типа масив(от цели числа, и не цели, дори и от низове).

#include <iostream>

using namespace std;

template <typename T, size\_t N>

void printArray(const T arr [], size\_t n) {

for (size\_t i = 0; i < n; ++i) {

cout << arr[i] << " ";

}

cout << endl;

}

int main() {

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

cout << "int arr: ";

printArray(intArr, 5);

double doubleArr[] = {1.3, 0.2, 7.56, 7.96, 8.26};

cout << "double arr: ";

printArray(doubleArr, 5);

return 0;

}