**Конспект: Область видимости. Передача параметров в функцию**

**Базовые понятия функции**

Все переменные имеют определенное время жизни (lifetime) и область видимости (scope). Время жизни начинается с момента определения переменной и длится до ее уничтожения. Область видимости представляет часть программы, в пределах которой можно использовать объект. Как правило, область видимости ограничивается блоком кода, который заключается в фигурные скобки. В зависимости от области видимости создаваемые объекты могут быть глобальными, локальными или автоматическими.

Пример:

void func1() { int a = 12; } // мы объявили переменную a. Мы можем её использовать в функции func1. Т.е. область видимости переменной а включает в себя эту функцию

void func2() { int b = a; } // мы объявили переменную b. Мы можем использовать её в функции func2, потому что это - её область видимости. Но область видимости переменной а не распространяется на эту функцию (другой блок кода), поэтому здесь мы получим ошибку.

int main() {

int c = a + b; // здесь мы объявили переменную с. Её область видимости - вся функция main. Но main не знает переменных a и b, потому что они были определены в других функциях. Ошибка!

}

**Глобальные переменные**

Глобальные переменные определены в файле программы вне любой из функций или любого другого блока кода и могут использоваться любой функцией. Глобальные переменные существуют в течение всей жизни программы и уничтожаются лишь с завершением программы.

Если глобальные переменные не инициализированы, то они получают нулевые значения.

Например, определим глобальную переменную n:

#include <iostream>

int n = 5; // глобальная переменная

void ndouble()

{

n \* 2;

std::cout << "n = " << n << std::endl;

}

int main()

{

ndouble(); // n=10

n++;

std::cout << "n = " << n << std::endl; // n=11

}

Здесь переменная n является глобальной и доступна из любой функции. При этом любая функция может изменить ее значение.

**Локальные переменные**

Переменные, которые создаются внутри определенного блока кода называются локальными. Такие объекты доступны в пределах только того блока кода, в котором они определены.

**Автоматические объекты**

Локальные переменные, которые существуют только во время выполнения того блока, в котором они определены, являются автоматическими.

Говорят, что такие объекты имеют автоматическую продолжительность хранения, или автоматическую протяженность. Неинициализированный автоматический объект содержит случайное, или неопределенное, значение, оставшееся от предыдущего использования памяти. После завершения функции память, в которой храниться переменная освобождается. Время жизни такого объекта заканчивается с завершением работы функции, и его значение теряется.

#include <iostream>

void print()

{

int n = 5; // локальная переменная, которая существует только в этой функции

std::cout << "n = " << n << std::endl;

}

Здесь в функции print определена локальная переменная n. Вне функции эта переменная недоступна. Например, мы не можем использовать переменную n в функции main, так как ее область видимости ограничена функцией print.

Подобным образом с помощью блока кода можно определить вложенные области видимости:

#include <iostream>

int main()

{

int n1 = 2; // область видимости - вся функция main

{

int n2 = 5; // область видимости - блок кода

std::cout << "n2 = " << n2 << std::endl;

n1++; // переменная n1 доступна, т.к. определена во внешнем контексте - функции

} // конец блока функции - конец времени жизни переменной n2

// переменная n1 доступна до конца функции

std::cout << "n1 = " << n1 << std::endl;

} // конец блока функции - конец времени жизни переменной n1

Для каждой области видимости доступны все те объекты, которые определены во внешней области видимости или во внешнем контексте. Глобальная область видимости является внешней для функции, поэтому функция может использовать глобальные переменные. А функция является внешним контекстом для вложенного блока кода, поэтому блок кода может использовать переменную n1, которая определена в функции вне этого блока. Однако переменные, определенные в блоке кода, вне этого блока использовать нельзя.

**Скрытие объектов**

Локальные объекты, определенные внутри одного контекста, могут скрывать объекты с тем же именем, определенные во внешнем контексте:

#include <iostream>

int n = 5;

int main()

{

int n = 10;

std::cout << "n = " << n << std::endl; // n=10

{

int n = 20;

std::cout << "n=" << n << std::endl; // n=20

}

}

Здесь определено три переменных с именем n. Переменная n, определенная на уровне функции main (int n = 10;) скрывает глобальную переменную n. А переменная n, определенная на уровне блока, скрывает переменную, определенную на уровне функции main.

Однако иногда бывает необходимо обратиться к глобальной переменной. В этом случае для обращения именно к глобальной переменной можно использовать оператор :: перед именем переменной

#include <iostream>

int n = 5;

int main()

{

int n = 10;

std::cout << "n=" << ::n << std::endl; // n=5

{

int n = 20;

std::cout << "n=" << ::n << std::endl; // n=5

}

}

**Передача параметров в функцию**

Обмен информацией между вызываемой и вызывающей функциями осуществляется с помощью механизма передачи параметров. Список переменных*,*указанный в заголовке функции называется формальными параметрамиили просто параметрами функции*.*

Передача параметров выполняется следующим образом. Вычисляются выражения, стоящие на месте фактических параметров. Затем формальным параметрам присваиваются значения фактических. Выполняется проверка типов и при необходимости выполняется их преобразование.

Передача параметров в функцию может осуществляться по значению и по адресу.

**Передача по значению**. Это когда наши аргументы не меняются при выходе из функции. В этом случае создаются копии переменных, которые мы передаём функции для использования. Первый пример - пример передачи по значению.

**Передача по адресу**. Все переменные, которые мы использовали в качестве фактических параметров, изменятся при выходе из функции. В этом случае мы работаем непосредственно с ячейками памяти, где хранятся значения переменных.

Пример:

#include <iostream>

using namespace std;

int func(int &a, int &b) //целочисленная функция по имени func, формальные параметры: &a, &b, т.е. мы работаем с адресами

{

a = a++; //выполняем тело функции

b += a;

return a-b; //и возвращаемся обратно

}

int main()

{

int x = 1;

int y = 2;

//Вывод х и у до функции

cout << "x before: " << x << endl;

cout << "y before: " << y << "\n\n";

//Вывод работы функции, х и у - фактические параметры

cout << "res: " << func(x, y) << endl;

//Вывод х и у после работы функции

cout << "x after: " << x << endl;

cout << "y after: " << y << endl;

}

**Упражнение**. Попробуй запустить код из примера. Запомни результаты. Теперь удали из кода амперсанды (&) и запусти опять. Сравни результаты с предыдущими.

**Константы в аргументах**. Когда мы передаём аргументы по ссылке мы можем изменить те данные, которые нельзя было менять. Для того, чтобы это не допустить, используются константы. Тогда мы точно не поменяем то, что нельзя менять.

Пример константы в аргументах:

int func(const int &a)

{

a++; //здесь будет ошибка, т.к. мы пытаемся изменить константу

return a;

}