**Область видимости объектов**

Ранее были рассмотрены  глобальные и локальные объекты в качестве переменных. Это базовый тип данных в котором сделать ошибку практически невозможно. Рассматривая тему функций нельзя говорить только о переменных, необходимо брать понятие в целом, поэтому в дальнейшем будет использоваться слово объект.

**Автоматические объекты**

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

При входе в блок для подобных переменных выделяется память, а после завершения работы этого блока, выделенная память освобождается, а объекты удаляются.

#include <iostream>

void print(int);

int main()

{

int z = 2;

print(z); // n=10

//n++; так сделать нельзя, так как n определена в функции print

return 0;

}

void print(int x)

{

int n = 5 \* x;

// z++; так сделать нельзя, так как z определена в функции main

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

}

Здесь в функции print определена локальная переменная n. В функции main определена автоматическая переменная z. Вне своих функций эти переменные недоступны. Например, мы не можем использовать переменную n в функции main, так как ее область видимости ограничена функцией print. Соответственно также мы не можем использовать переменную z в функции print, так как эта переменная ограничена фукцией main.

Параметры функции также, как и локальные переменные, существуют, пока выполняется функция, вне функции они не доступны.

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

#include <iostream>

int main()

{

int n = 2;

{

int x = 5;

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

n++; // так можно, так как n определена во внешнем контексте

}

//x++; // так сделать нельзя, так как x определена в блоке кода

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

return 0;

}

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 n, которая определена в функции вне этого блока. Однако переменные, определенные в блоке кода, вне этого блока использовать нельзя.

**Сокрытие 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

}

return 0;

}

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.

**Статические объекты**

Кроме автоматических есть особый тип локальных объектов - статические объекты. Они определяются на уровне функций с помощью ключевого слова static. Если автоматические переменные определяются и инициализируются при каждом входе в функцию, то статические переменные инициализируются только один раз, а при последующих вызовах функции используется старое значение статической переменной.

Например, пусть будет функция со стандартной автоматической переменной:

#include <iostream>

void display();

int main()

{

display();

display();

display();

return 0;

}

void display()

{

int i = 0;

i++;

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

}

Функция display вызывается три раза, и при каждом вызове программа повторно будет выделять память для переменной i, которая определена в функции. А после завершения работы display, память для переменной i будет освобождаться. Соответственно ее значение при каждом вызове будет неизменно.

Если сделать переменную i статической, то при завершении работы функции display переменная не уничтожается, ее память не очищается, наоборот, она сохраняется в памяти. И соответственно результат работы программы будет иным.

**Разделение программы на файлы**

При разработке программ рекомендуется разбивать их на части, которые функционально ограничены и закончены. Например, некоторые функции можно расположить в  отдельных \*.сpp файлах. Такой подход обеспечивает ряд преимуществ:

* Обычно сложная программа разбивается на несколько отдельных частей (модулей), которые отлаживаются отдельно и зачастую разными людьми; поэтому в завершении остается лишь собрать готовые модули в единый проект;
* При исправлении в одном модуле не надо снова транслировать (переводить в машинные коды) все остальные (это могут быть десятки тысяч строк);
* При компоновке во многих системах можно подключать модули, написанные на  других языках, например, на C# (в машинных кодах).

Например, определим файл factorial.cpp, который будет иметь следующий код:

int factorial(int n)

{

if (n > 1)

return n \* factorial(n - 1);

return 1;

}

Это функция вычисления факториала.

Добавим еще один файл factorial.h, который будет содержать объявление функции factorial:

int factorial(int);

Если ваш заголовочный файл может быть использован более одного раза, рекомендуется использовать директиву #pragma once. Это нестандартная, но широко распространённая препроцессорная директива, разработанная для контроля за тем, чтобы конкретный исходный файл при компиляции подключался строго один раз.

И также определим главный файл, который назовем main.cpp:

#include <iostream>

#include "factorial.h"

int main()

{

int result = factorial(5);

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

return 0; }

Функция main вызывает функцию factorial для вычисления факториала числа. Но чтобы использовать функцию, определенную в другом файле, необходимо добавить ее объявление. В прошлых темах объявление функции добавлялось непосредственно в главный и единственный файл программы. Однако если функции определены в отдельных файлах, то более оптимально помещать объявления функций в специальные заголовочные файлы и потом подключать эти файлы. Именно поэтому в начале с помощью директивы include подключается файл factorial.h, который содержит объявление или заголовок функции. Файл factorial.h еще называется заголовочным файлом (header file).

Можно было бы и не подключать файл factorial.h и вообще не создавать его, а объявление функции поместить непосредственно в файл main.cpp. Но при изменении функции может потребоваться изменить и ее объявление. И если функция factorial используется в нескольких файлах с исходным кодом, то в каждом из этих файлов придется менять ее объявление. В данном же случае достаточно изменить объявление функции в одном файле - factorial.h.

При компиляции в Visual Studio все файлы автоматически компилируются в один. При компиляции через g++ необходимо передать все файлы через пробел компилятору: g++ app.cpp factorial.cpp -o main

На выходе будет сгенерирован единый файл main.

**Внешние объекты**

Кроме функций внешние файлы могут содержать различные объекты - переменные и константы. Для подключения внешних объектов в файл кода применяется ключевое слово extern.

Для объявления объектов определим файл objects.h со следующим содержимым:

extern const int x;

extern double y;

Здесь объявляются константа x и переменная y. Так как эти объекты будут внешними по отношению к исходному коду, который будет их использовать, то они определяются с ключевым словом extern.

Для определения этих объектов добавим новый файл objects.cpp:

#include "objects.h"

const int x = 5;

double y = 3.4;

Используем эти объекты в файле main.cpp:

#include <iostream>

#include "objects.h"

int main() {

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

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

return 0;

}