Модуль 5

- Область видимости переменных
- Глобальные переменные
- Статические переменные
- Пространства имен
- Перечисления
- Преобразование типов
  auto, decltype и вывод типов

Область видимости переменных

Область видимости определяет, где можно использовать переменную.

Продолжительность жизни (или ***«время жизни»***) определяет, где переменная создается и где уничтожается. Эти две концепции связаны между собой.

```
{

    // локальные переменные



}
```

Область видимости переменных

```
#include <iostream>
// Параметр x можно использовать только внутри функции add()
int add(int x, int y) // параметр x функции add() создается здесь
{
    return x + y;
} // параметр x функции add() уничтожается здесь
// Переменную x функции main() можно использовать только внутри функции main()
int main()
{
    int x = 5; // переменная x функции main() создается здесь
    int y = 6;
    std::cout << add(x, y) << std::endl; // значение x функции main() копируется в
переменную x функции add()
    return 0;
} // переменная x функции main() уничтожается здесь
```

Область видимости переменных

Сокрытием имен - Переменная внутри вложенного блока может иметь то же имя, что и переменная внутри внешнего блока

Избегайте использования вложенных переменных с именами, идентичными именам внешних переменных!!

Область видимости переменных

Определяйте переменные в наиболее ограниченной области видимости.

Область видимости переменных

Параметры функций имеют локальную область видимости

```
int max(int x, int y) // x и y определяются здесь
{
    // Присваиваем большее из значений (x или y) переменной max
    int max = (x > y) ? x : y; // max определяется здесь
    return max;
} // x, y и max уничтожаются здесь
```

Глобальные переменные

Глобальными называются переменные, которые объявлены вне блока. Они имеют **статическую продолжительность жизни**, т.е. создаются при запуске программы и уничтожаются при её завершении.

Использовать одинаковые имена для локальных и глобальных переменных — это прямой путь к проблемам и ошибкам!!!

Ключевое слово static

ключевое слово static необходимо использовать, если хотите, чтобы эта переменная была доступна только в этом конкретном файле.

```
#include <iostream>

static int g_x; // g_x - это статическая глобальная переменная,
которую можно использовать только внутри этого файла
int main()
{
    return 0;
}
```

Ключевое слово extern

Чтобы использовать внешнюю глобальную переменную, которая была объявлена в другом файле, нужно записать предварительное объявление переменной с использованием ключевого слова `extern.`

Связывание функций

Функции имеют такие же свойства связи, что и переменные. По умолчанию они имеют внешнюю связь, которую можно сменить на внутреннюю с помощью ключевого слова `static`.

Предварительные объявления функций не нуждаются в ключевом слове `extern`. Компилятор может определить сам (по телу функции): определяете ли вы функцию или пишете её прототип.

Глобальные переменные - ЗЛО!!

- **Какой наилучший префикс для глобальных переменных?**
- **//**

1. Использование глобальных не константных переменных увеличивает количество мест с потенциальным обращением к глобальной переменной, что приводит к усложнению отладки кода. Используйте инициализацию локальных переменных как можно ближе к месту использования.
2. Есть возможность изменения значения в коде, что запутает разработчика при понимании логики программы.
3. Уменьшается модульность приложения.

Статические переменные

Использование **ключевого слова static** с локальными переменными изменяет их свойство продолжительности жизни с автоматического на статическое (или *«фиксированное»*). **Статическая переменная** (или ***«переменная со статической продолжительностью жизни»***) сохраняет свое значение даже после выхода из блока, в котором она определена.

```
int generateID()
{
    static int s_itemID = 0;
    return s_itemID++;
}
```

Пространства имен

Конфликт имен возникает, когда два одинаковых идентификатора находятся в одной области видимости, и компилятор не может понять, какой из этих двух следует использовать в конкретной ситуации.

Для решения подобных проблем и добавили в язык C++ такую концепцию, как **пространства имен**.

Пространство имен определяет область кода, в которой гарантируется уникальность всех идентификаторов.

Пространства имен

using-объявления

```
#include <iostream>
int main()
{
   using std::cout; // "using-объявление" сообщает компилятору, что cout следует обрабатывать, как std::cout
   cout << "Hello, world!"; // и никакого префикса std:: уже здесь не нужно!
   return 0;
}
```

Пространства имен

using-директивы

```
#include <iostream>
int main()
{
   using namespace std; // "using-директива" сообщает компилятору, что мы используем все объекты из пространства имен std!
   cout << "Hello, world!"; // так что никакого префикса std:: здесь уже не нужно!
   return 0;
}
```

Пространства имен

Если «**using-объявление**» или «**using-директива**» используются в блоке, то они применяются только внутри этого блока (по обычным правилам локальной области видимости). Это хорошо, поскольку уменьшает масштабы возникновения конфликтов имен до отдельных блоков.

Старайтесь не использовать “*using*” вне тела функций.

Пространства имен

Как только один “***using***” был объявлен, его невозможно отменить или заменить другим using-стейтментом в пределах области видимости, в которой он был объявлен.

```
int main()
{
    {
        using namespace Boo;
        // Здесь всё относится к пространству имен Boo::
    } // действие using namespace Boo заканчивается здесь
    {
        using namespace Foo;
        // Здесь всё относится к пространству имен Foo::
    } // действие using namespace Foo заканчивается здесь
    return 0;
}
```

Перечисления

Перечисление (или ***«перечисляемый тип»***) — это тип данных, где любое значение (или ***«перечислитель»***) определяется как символьная константа.

Перечисления

```
// Объявляем новое перечисление Colors
enum Colors
{
    // Ниже находятся перечислители - все возможные значения этого типа данных.
    // Каждый перечислитель отделяется запятой (НЕ точкой с запятой)
    COLOR_RED,
    COLOR_BROWN,
    COLOR_GRAY,
    COLOR_WHITE,
    COLOR_PINK,
    COLOR_ORANGE,
    COLOR_BLUE,
    COLOR_PURPLE,
}; // enum должен заканчиваться точкой с запятой

// Определяем несколько переменных перечисляемого типа Colors
Colors paint = COLOR_RED;
Colors house(COLOR_GRAY);
```

Перечисления. Польза.

```
enum ParseResult
{
    SUCCESS = 0,
    ERROR_OPENING_FILE = -1,
    ERROR_PARSING_FILE = -2,
    ERROR_READING_FILE = -3
};
ParseResult readFileContents()
{
    if (!openFile())
        return ERROR_OPENING_FILE;
    if (!parseFile())
        return ERROR_PARSING_FILE;
    if (!readfile())
        return ERROR_READING_FILE;
    return SUCCESS; // если всё прошло успешно
}
```

Преобразование типов auto, decltype и вывод типов

Начиная с C++11 **ключевое слово auto** при инициализации переменной может использоваться вместо типа переменной,

```
auto x = 4.0; // 4.0 - это литерал типа double, поэтому и переменная x должна быть типа double
auto y = 3 + 4;
```

```
auto subtract(int a, int b)
{
    return a - b;
}
```

В параметрах функции должен быть указан тип явно, возвращаемое значение может быть auto, но делать так не стоит!

decltype

decltype - это оператор в C++, который возвращает тип выражения или значения. Он позволяет получить тип переменной или выражения без вычисления.

```
int x = 5;
decltype(x) y; // y будет иметь тип int, так как x имеет тип int


int foo() { return 0; }
decltype(foo()) result;
```