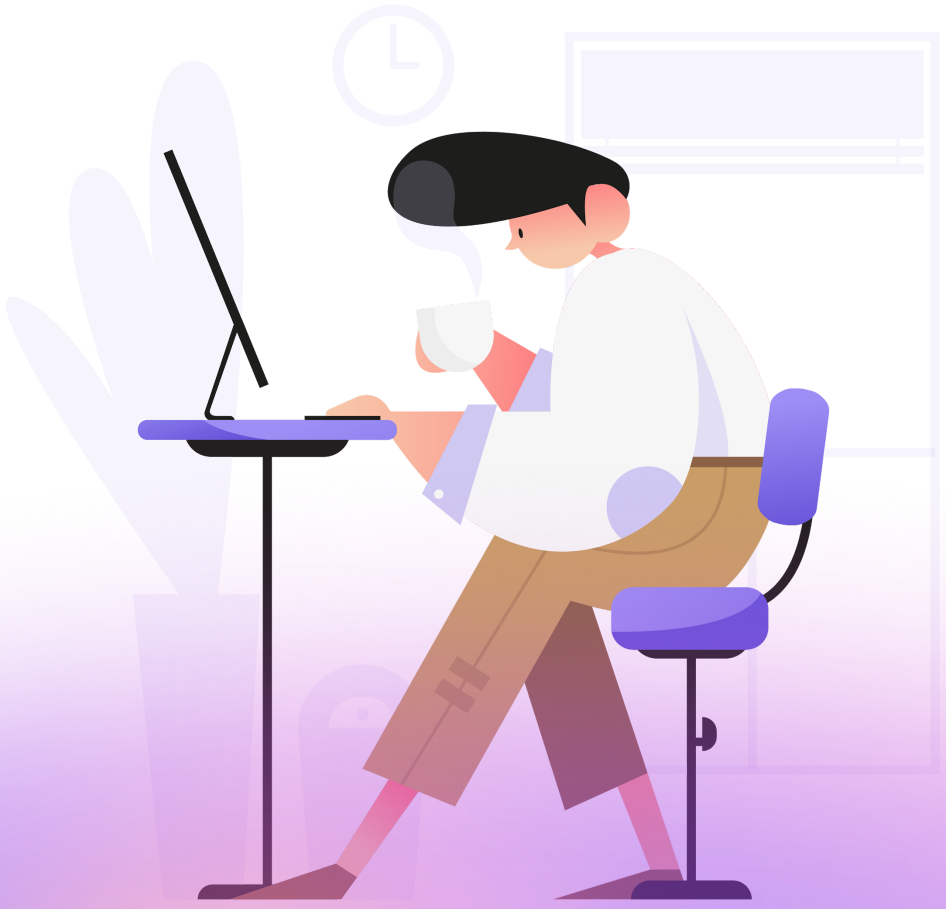
GeekBrains

Основы C++

Основные понятия



На этом уроке

1. Изучим понятие переменных и типов данных.
2. Рассмотрим классы памяти, области действия и время жизни переменных.
3. Научимся использовать такие типы данных как массивы, структуры, объединения.
4. Узнаем, как происходит индексация и что такое арифметика указателей.

Оглавление

На этом уроке

Переменные и типы данных.

Операнды в C++

- Названия
- Тип данных.
- Значение переменной
- Класс памяти

Область видимости

Время жизни

Квалификаторы типов

- Переменные перечисляемых типов
- Переопределение типов переменных

Массивы и структуры. Индексация и арифметика указателей.

- Многомерные массивы.

Тип данных структуры

- Массивы структур.

Объединения (union)

- Структуры внутри структур
- Битовые поля

Практическое задание

Дополнительные материалы

Используемые источники

Переменные и типы данных.

На прошлом занятии мы поговорили о понятиях функции, оператора, выражения, и лишь слегка коснулись понятия данных. Такой подход связан с тем, что языки C/C++ от самого своего зарождения опирался на модель вычислительной машины фон-Неймана и вначале был предназначен для написания операционной системы (ОС) UNIX. Матричные процессоры, нейронные сети, облачные вычисления появились гораздо позже, но и они подвластны C++, а вычислитель фон-Неймана устроен, в общих чертах, так: существует поток данных, который надо принять по шине данных из памяти или с устройств ввода, обработать по нужному алгоритму, и вывести результаты на устройства вывода или в память. Для реализации алгоритмов обработки данных другой поток - поток команд - запускается из памяти машины через шину команд. Именно поток команд является главным, он управляет, в том числе, началом и завершением, а также очередностью выбора данных для обработки. Так и в программе на языке C++ все начинается с вызова функции `main()`, явного вызова, как в UNIX, или немного спрятанного, как в Windows, это не важно. Главное - поток команд управляет потоком данных.

Можно заметить, что в живой природе, да и в технике, все происходит как раз наоборот: принимая через наши органы чувств информацию, мы формируем в своем мозгу ответную реакцию на поступающую информацию. На этом уроке поговорим, как раз, о потоке данных.

В языке C++ данные принято называть **операндами** (то есть “то, над чем совершаются операции”), и делить операнды на два базовых типа: константы и переменные. Константы принимают свои значения перед выполнением программы и сохраняют их до завершения работы программы. Переменные могут изменять свои значения в процессе выполнения программы.

Операнды в C++

Имеют четыре главных атрибута, а также квалификатор типа. Атрибуты:

1. Название;
2. Тип данных;
3. Значение;
4. Класс памяти.

Названия

Строго говоря, символические имена, используются для обращения к константам и переменным. Выбирая имена, программист должен соблюдать пять основных правил:

1. имя должно состоять только из букв латинского алфавита, цифр и символов подчеркивания;
2. имя должно начинаться с буквы, хоть некоторые компиляторы и допускают в этом вопросе произвол, но начинать имя с цифры - дурной стиль, всегда можно вставить цифру в середину или конец имени (подробнее о признаке числа - на прошлом уроке);

3. максимальный размер имени 255 значащих символов, почти все компиляторы просто игнорируют все символы после 255-го, ошибки не будет, если имена различаются в первых 255 символах;
4. прописные и строчные буквы различаются;
5. нельзя использовать в качестве имени зарезервированные слова. В начале этого курса не зря упоминались стандарты ISO/IEC, полный список зарезервированных слов приведен там, он может обновляться, и всегда надо сверяться с последней редакцией стандарта. Конечно, компиляторы на которых проходит обучение и на которых работает подавляющее большинство предприятий отстают от последних версий стандарта лет на пять. Но это не повод опираться на устаревшую информацию. Вот, например, такие списки из черновика (working draft) стандарта ISO/IEC 14882:2019:

```
aligns      constexpr     extern     private           this
alignof     const_cast    false      protected         thread
asm         continue      float      public            throw
auto        co_await      for        register          true
bool        co_return     friend     reinterpret_cast  try
break       co_yield      goto       requires          typedef
case        decltype      if         return            typeid
catch       default       inline     short             typename
char        delete        int        signed            union
char8_t     do            long       sizeof            unsigned
char16_t    double        mutable    static            using
char32_t    dynamic_cast  namespace  static_assert     virtual
class       else          new        static_cast       void
concept     enum          noexcept   struct            volatile
const       explicit      nullptr    switch            wchar_t
consteval   export        operator   template          while
and  and_eq  bitand  bitor  compl  not  not_eq  or  or_eq  xor  xor_eq
```

Тип данных.

Обязательный атрибут для констант и переменных. Компилятору необходимо знать тип данных, чтобы выделить необходимый размер памяти для хранения константы или переменной, и затем правильно интерпретировать содержимое выделенной памяти. Стандартные типы данных согласно ISO/IEC:

Suffix	Decimal literal	Binary, octal or hexadecimal literal
-	int long int long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u or U	unsigned int unsigned long int	unsigned int unsigned long int

	`unsigned long long int`	`unsigned long long int` `long int`

`l or L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`Both u or U and l or L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll or LL`	`long long int`	`long long int` `unsigned long long int`
`Both u or U and ll or LL`	`unsigned long long int`	`unsigned long long int`

Это означает, что:

- существуют базовые типы данных:
  - целые: int, long, short, unsigned;
  - символьные: char;
  - с плавающей запятой: float, double;
- базовые типы могут использоваться как самостоятельно, например: `int a = 8;` здесь переменной по имени `a` типа `int` присваивается начальное значение `8` десятичное;
- так и комбинироваться с суффиксами или литералами, в качестве которых выступают те же литералы базовых типов: `uchar b = 'B';` здесь беззнаковая символьная переменная по имени `b` инициализируется начальным значением 'символ B'. В этом примере литерал типа `char` дополняется суффиксом `u`, что означает `unsigned`, то есть, беззнаковый (без такого суффикса, по умолчанию, тип `char` является знаковым). `unsigned char b = 'B';` такая запись эквивалентна предыдущей, но вместо суффикса `u` использован литерал `unsigned`.

Без ответа остается вопрос: так сколько байт займет на самом деле в памяти компьютера переменная из вышеприведенного примера? Сколько байт компилятор отведет под переменную типа `long double` на конкретном компьютере?

В стандартной библиотеке C есть заголовочный файл **`stdint.h`**, введённый стандартом ISO/IEC 9899:1999 (известным также как C99). Заголовочный файл объявляет целочисленные типы, которые имеют заданный размер и являются наиболее быстрыми при использовании. В дополнение к ним стандарт объявляет макросы, устанавливающие размер этих типов. Вот фрагменты этого файла:

```
typedef signed   char          int8_t;
typedef unsigned char          uint8_t;
typedef signed   short         int16_t;
typedef unsigned short         uint16_t;
typedef signed   long          int32_t;
typedef unsigned long          uint32_t;
```

```
#define INT8_MIN                (-INT8_MAX-1)
#define INT8_MAX                (0x7f)
#define UINT8_MAX               (0xffU)
#define INT16_MIN               (-INT16_MAX-1)
#define INT16_MAX               (0x7fff)
#define UINT16_MAX              (0xffffU)
#define INT32_MIN               (-INT32_MAX-1)
#define INT32_MAX               (0x7fffffffL)
#define UINT32_MAX              (0xffffffffUL)
```

Чтобы получить имя, представляющее минимальное или максимальное значение данного типа, возьмите имя типа, замените `_t` на `_MIN` или `_MAX` и переведите все символы в верхний регистр. Например, наименьшим значением для типа `int32_t` является `INT32_MIN`.

Существует также специальная операция `sizeof`, которая используется для определения размера памяти в байтах, занимаемой тем или иным типом данных именно на конечном компьютере, на котором исполняется программа:

`sizeof(int);` вернет значение 4 на 32- разрядной платформе;

`sizeof(double);` скорее всего вернет 8 на той- же платформе.
Рекомендуем перед началом работы со сложными типами данных вроде массивов и структур, о которых пойдет речь далее, уточнить каковы истинные размеры базовых типов переменных на вашем компьютере.

```
printf("%d\n", sizeof(int));
printf("%d\n", sizeof(char));
printf("%d\n", sizeof(long));
printf("%d\n", sizeof(float));
```

Значение переменной

Мы уже показали присвоение значения переменной при инициализации переменной чуть выше. Здесь только одно замечание. Инициализация переменных устанавливает значение переменных во время компиляции, а не во время выполнения программы. Чтобы установить значение переменной во время выполнения программы, нужно использовать операцию присваивания, которая будет рассмотрена позднее.

```
int x, y;          // переменные x, y объявлены, но не проинициализированы.
                   // Вполне допустимо.
int x, int y;      // Ошибка! Запятыми должны разделяются имена
                   // переменных, как показано выше, а не их объявления целиком
char symb, let = 'A', num = '5';  // Корректно. Переменная symb только
                                  // объявлена, а переменные let и num того же
                                  // типа объявлены и проинициализированы.
char c = '\033';   // Здесь переменная инициализируется не отображаемым
                   // специальным символом ESCAPE. Обратите внимание
                    // на обратный слеш перед его ASCII- кодом.
```

```
char NewLine = '\n';  // Переменная инициализируется специальным
                      //символом перевода строки.
```

Если мы объявляем переменную, но не инициализируем её значение сразу, то компилятор не гарантирует нам никакое значение этой переменной по умолчанию. Да, велика вероятность, что значением такой переменной будет ноль, но гарантировать это - нельзя.

Класс памяти

В языке C++ любая область памяти компьютера, которая может быть использована программой, называется объектом (на данном этапе - не путать с объектами объектно-ориентированного программирования). Любое выражение, представляющее собой ссылку на объект, называется адресным. Например, в выражении int min = 0; идентификатор min представляет собой ссылку на объект, способный содержать целое число нуль, т.е. адрес объекта. Так же и имя функции является просто идентификатором. В дополнение к имени, типу и значению объекта существуют еще два атрибута: область действия и время жизни. Эти два атрибута определяются классом хранения (известным также под именем класс памяти), который связывается с конкретным объектом.
Мы еще не рассматривали во всех подробностях функции, а тем более понятия объектов в диалекте языка C++, поэтому рассмотрим здесь понятие классов памяти в узком смысле, применительно к переменным. Однако будем иметь в виду, что излагаемый материал применим и к другим объектам языка C/C++. Механика классов памяти в языке C работает согласно таблицы:

Класс памяти	Тип переменной	Область действия	Время жизни
`auto (register)`	внутренняя	Функция или блок кода	Функция или блок кода
`extern`	внешняя	Остаток файла, другие файлы с объявлением `extern`	Программа
`static`	внешняя	Остаток файла (только внутри файла)	Программа
	внутренняя	Функция или блок кода	
`(опущен)`	внешняя	Как `extern`	Как `extern`
	внутренняя	Как `auto`	Как `auto`

Язык Си позволяет программисту управлять выделением области памяти для хранения данного объекта. (Сейчас речь идет о статическом распределении памяти, которое производится на этапе компиляции программы и сохраняется неизменным вплоть до завершения программы. О динамическом выделении памяти поговорим дополнительно на одном из следующих занятий, специально посвященных этой теме). Программа на языке C++, по большому счету, состоит из трех

сегментов: команд, данных и так называемого стека[1] (стека вызовов). Объекты данных могут храниться в сегменте как данных, так и стека. Это определяется классом хранения объекта, который в свою очередь задает время жизни объекта и его область действия. В языке C++ предусмотрены пять классов хранения: внешний (`extern`), статические (`static`) внешний и внутренний, автоматический (`auto`, не путать с ключевым словом `auto`, описанным в стандарте C++11) и регистровый (`register`). Все они показаны выше в таблице.

Область видимости

Областью действия (синоним: областью видимости) переменной называется та часть программы, в которой можно пользоваться этой переменной. Язык C++ позволяет задавать область видимости каждого элемента данных и область действия каждой функции. Временем жизни объекта данных называется отрезок времени исполнения программы, в течение которого значение этого объекта доступно для использования в некоторой части программы. Время жизни объекта может быть столь коротким, как время исполнения операторов блока, или столь же длинным, как время выполнения всей программы.

Вначале рассмотрим внешний (`extern`) и внешний статический (`static`) классы. Для этого детальнее ознакомимся с синтаксисом модулей в языке C++. В частности, можно объявлять и определять типы данных в модуле до того, как будут указаны определения функций. В этом и состоит смысл термина "внешнее определение данных": данные определяются внешним образом по отношению к функциям. Рассмотрим следующий модуль (обращайте внимание на синтаксис, но подробнее об операциях и функциях мы еще поговорим на следующих занятиях):

```
int val_1, val_2; // Определить внешние данные
void save() {     // Эта функция  сохраняет некоторые  значения
    val_1 = 10;
    val_2 = 15;
    return;
}
```

---

[1] Примечание (из Википедии): Стек (англ. stack — стопка) — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»). Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю. Или с магазином в огнестрельном оружии (стрельба начнётся с патрона, заряженного последним). В 1946 Алан Тьюринг ввёл понятие стека. А в 1957 году немцы Клаус Самельсон и Фридрих Л. Бауэр запатентовали идею Тьюринга.
В языке C++ стандартная библиотека имеет класс `stack` с реализованной структурой и методами.
Для нас в программах и в данном контексте важно, что это область памяти, с которой непосредственно проводит манипуляции центральный процессор, формируется он действительно по принципу очень близкому к LIFO и является эта область наиболее динамичной, туда попадают все локальные переменные вызова функции и производятся вычисления

```
int main()   {
    save();
    printf("Значения равны:   %d  %d\n",val_1, val_2);
}
```

В этом примере определены два внешних объекта данных: `val_1` и `val_2`. Эти объекты будут размещены в сегменте данных программы. Область действия внешних объектов данных распространяется на весь модуль, в таком случае говорят, что они глобальные. Любая функция, находящаяся в этом файле (модуле), может иметь доступ к внешним (глобальным) объектам данных. Таким образом, переменные `val_1` и `val_2` являются общими для функций `main()` и `save()`. Область действия внешних данных можно распространить и за пределы модуля, используя служебное слово `extern` (external = внешний). В предыдущем примере распределим две функции по разным исходным файлам (модулям):

Модуль **main.c**	Модуль **save.c**

`extern int val_1;`
`extern int val_2;`
`void save();`
`int main()   {`
`    save();`
`    printf("Значения равны: %d %d\n", val_1, val_2);`
`    return;`
`}`
	`int val_1;`
	`int val_2;`
	
	`void save() {`
	`    val_1 = 10;`
	`    val_2 = 15;`
	`}`

Данные *определены* в модуле **save.c**, а в модуле **main.c** они *объявлены* с помощью служебного слова `extern`.

***Внимание! Определение* создает объект данных, а *объявление* - ссылку на этот объект.**

При разделении программы на несколько файлов важно помнить, что язык требует явного указания используемых в программе функций, поэтому файлу **main.c** необходимо явно указать на то, что в нём содержится вызов функции из файла **save.c**. Также важно, если Вы используете макросборщик указать все файлы сборщику в строке `add_executable(main src/main.c src/save.c)`

Указание перед спецификацией типа данных служебного слова `static` (статический) принудительно ограничивает область действия объекта данных только тем модулем, в котором он определен. Рассмотрим пример:

Модуль **main.c**	Модуль **save.c**

`extern int val_1;`
`extern int val_2;`
`void save();`
`int main()   {`
`    save();`
`    printf("Значения равны: %d %d\n", val_1, val_2);`
`    return;`
	`static int val_1;`
	`int val_2;`
	
	`void save() {`
	`    val_1 = 10;`
	`    val_2 = 15;`
	`}`

}	

В модуле **save.c** теперь `static int val_1;` с ограниченной областью действия. Так как функция `main` все еще ссылается на `val_1`, то при попытке компиляции и компоновки этих модулей, во время загрузки, будет обнаружена примерно такая (в зависимости от конкретного компоновщика и загрузчика) ошибка:

```
 Unresolved Externals:
 VAL_1 in file(s):
 MAIN.OBJ(MAIN)
 There was 1 error detected
```

Линковщик в составе транслятора GCC выдаёт схожую по смыслу ошибку

```
Undefined symbols for architecture x86_64:
  "_val1", referenced from:
      _main in main.cpp.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1
```

Т.е. загрузчик не информирован о существовании имени `val_1`. Когда он начал поиск этого имени (при попытке разрешить ссылку на имя `val_1` в функции `main`), то он не нашел его, поскольку область действия этого имени теперь ограничена модулем **save.c**. Не сумев установить требуемую связь между определением переменной `val_1` в модуле **save.c** и ссылкой на это имя в модуле **main.c**, загрузчик и выдал сообщение **Unresolved (Undefined) Externals (symbols)**.
Определение функций и переменных статическими, где это возможно, относится к хорошему стилю разработки программ, поскольку дает возможность избежать конфликта имен, т.е. ситуации, когда имена объектов в разных модулях совпадают, что вызывает ошибки как на этапе компиляции, так и на этапе исполнения программы. Снова обращаем ваше внимание на то, что все сведения о ключевых словах на данный момент приводятся в контексте процедурного стиля программирования, и в контексте объектно-ориентированного программирования ключевые слова (например `static`) могут и будут обладать дополнительными свойствами.

Время жизни

Временем жизни внешних данных является время жизни всей программы. Это верно как для внешних, так и для внешних статических данных. Они хранятся в сегменте статических данных программы, поскольку не должны быть утрачены между вызовами функций. Однако по своей области действия внешние статические данные существуют только для модуля, в котором они определены.
Определения и объявления, входящие только в состав блока кода, называются внутренними (или локальными) определениями и объявлениями. Они имеют тот же синтаксис, что и внешние определения и объявления. Локальные объекты данных привязаны к отдельным функциям или блокам. Данные могут быть определены внутри блока как имеющие либо автоматический (`auto`), либо

статический (`static`), либо регистровый (`register`) класс хранения. Это задается с помощью одного из служебных слов (`auto`, `static`, или `register`). По умолчанию (если служебное слово опущено) подразумевается класс `auto`. Областью действия объектов, определенных внутри блока, является этот блок или любые вложенные в него блоки.

Внутренние статические объекты размещаются в сегменте данных. Они являются постоянными (не путать с константными значениями), и их время жизни совпадает с временем жизни всей программы. Статические объекты создаются однократно и сохраняют свои значения между повторными входами в блок, в котором они определены.

Внутренние автоматические (`auto`) – это объекты с автоматическим классом хранения.

Автоматические объекты, наряду с регистровыми и внутренними статическими, могут быть определены внутри любого блока операторов языка C++. Автоматические объекты данных запоминаются в сегменте стека. Следовательно, они создаются при входе в блок и уничтожаются при выходе из него. Тем самым их время жизни совпадает с временем исполнения блока.

Служебное слово `register` (регистровый), определяющее класс хранения, указывает компилятору, что надо попытаться хранить соответствующий объект в машинных регистрах (если это возможно). Регистры - это самый быстродействующий вид памяти компьютера, он иногда используется для хранения данных, к которым требуется многократный доступ, например счетчиков. Так как у центрального процессора число регистров обычно невелико, то определение объекта как регистрового не гарантирует, что для его хранения будет выделен именно регистр. Если свободных регистров нет, то такой объект будет создан как автоматический. Регистровые объекты имеют то же время жизни, что и автоматические.

Квалификаторы типов

С помощью квалификатора типа компилятор C++ предоставляет программисту механизм, позволяющий влиять на эффективность кода, сгенерированного компилятором.

Квалификатор `const` определяет переменную, значение которой никак не может быть изменено во время выполнения программы. Поэтому такая переменная должна быть обязательно инициализирована при объявлении.

Квалификатор `volatile` определяет переменную, с которой компилятору запрещено проводить какие-либо оптимизирующие действия.

```
const float pi = 3.1415926;
volatile char port;
pi = 3.14;   // Ошибка!
port = 'Y';
port = 'N';
const x = 10; // На большинстве платформ допустимо. Компилятор присвоит
              // переменным x, y тип int по умолчанию.
volatile y;   // А переменную с квалификатором volatile
              // допускается не инициализировать.
```

Переменные перечисляемых типов

Переменная перечисляемого типа может иметь только одно из перечисленных при ее определении значений. Определяется перечисляемый тип при помощи ключевого слова `enum`. Например, запись `enum logi {false, true};` создает переменную `logi`, которая может принимать только одно из двух значений: `true` или `false`. Такие перечисления (с указанием слов истина и ложь) были чрезвычайно популярны в языке С, поскольку в нём нет булева типа, и ключевые слова `true`, `false` не были определены, а оперировать цифрами `1` и `0` бывает не очень удобно.

Компилятор каждому из списка перечисленных значений ставит в соответствие константу типа `int`, начиная с нуля, следовательно: `false = 0, true = 1`.

```
enum week {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
        // Неделя в американском стиле, начинается с воскресенья.
        // Элементам данного списка компилятор присвоит следующие
        // значения Sun=0; Mon=1; Tue=2; Wed=3; Thu=4; Fri=5; Sat=6;
enum week Today, Tomorrow; // Объявляем переменные только что объявленного
                           // перечисляемого типа
Today = Tue;               // Инициализируем объявленные переменные.
                           // Переменной Today будет
Tomorrow = Wed;            // присвоено значение 2;
                           // переменной Tomorrow значение 3.
Today = 2; // Также допустимое присвоение
Today = 7; // Ошибка, такого значения в перечислении нет!
```

Возможна и другая форма задания списка значений:

```
enum diary {Sun = 5, Mon, Tue, Thu, Fri, Sat}; // Элементам данного списка
                                               // будут присвоены значения:
enum diary Today, Tomorrow;   // Sun=5; Mon=6; Tue=7; Thu=8; Fri=9; Sat=10;
Today = 7;                    // В данном случае будет корректно
Today = 2;                    // Ошибка!
Tomorrow = Wed;               // Ошибка! (такого значения нет в перечислении)
```

Переопределение типов переменных

Программист может назначить дополнительное имя любому из типов при помощи инструкции

```
typedef старый_тип новый_тип;
```

Например:

```
typedef enum logi {fls, tr} LOGICAL;
LOGICAL flag;  // Так ранее приведенный пример становится
               // более удобным для восприятия,
flag = tr;     // и вместе с тем запись logi flg = tr;
               // остается корректной
```

Массивы и структуры. Индексация и арифметика указателей.

Массивы являются очень удобным инструментом языка, применяемым во многих программах. Массив предназначен для размещения большого количества переменных, имеющих один и тот же тип, под одним единственным именем. Массив может быть образован из данных любого типа. Имя массива, на самом деле, является его адресом, точнее адресом его нулевого элемента. Имя массива не является `lvalue`, Но один элемент массива является `lvalue`.

Примечание: в контексте наших занятий слушателям следует понимать термины `lvalue` и `rvalue` в упрощённой форме. `lvalue` (locator value) представляет собой объект, который занимает идентифицируемое место в памяти (например, имеет адрес, проще говоря, находится слева от оператора присваивания и является контейнером для хранения значений). `rvalue` это выражение, которое не представляет собой объект, занимающий идентифицируемое место в памяти (проще говоря, он не может стоять слева от оператора присваивания `=`, только справа и является значением, которое присваивается переменной).

Внимание! Нумерация элементов в массиве всегда начинается с нуля, а не с единицы, и все элементы массива должны иметь один и тот же тип. Игнорирование любого из этих правил неизбежно приводит к ошибке при работе с массивами.

Массив должен быть объявлен перед тем, как его элементы будут использоваться в программе. Пример объявления одномерного массива:

```
int data[5]; // Массив по имени data из 5 переменных типа int
```

Обратиться к элементу массива можно одним из двух способов:

- указав имя массива и индекс необходимого элемента;
- по указателю на необходимый элемент массива.

Индекс каждого элемента массива является не чем иным, как номером этого элемента в массиве. Еще раз напомним: нумерация начинается с нуля. Пример инициализации (присвоения значений всем элементам) вышеприведенного массива:

```
data[0]=5;
data[1]=87;
data[2]=5;
data[3]=0;
data[4]=125;
```

Типичные ошибки при такой инициализации:

```
data = 5; // Ошибка! data - это адрес массива.
data++;   // Ошибка по той- же причине. Имя массива не является lvalue
```

Но вполне допустимо:

```
data[i]=5;    // где i как раз и есть "индекс необходимого элемента".
data[i]++;    // быть присвоено необходимое значение, например, int i = 3;
data[i++]++; // Выражение также допустимо. Подумайте, что оно значит.
```

Элемент массива может быть использован в вычислениях везде, где может быть использована простая переменная того же типа. Например `sum = data[1] + data[4];`

Как заметит внимательный слушатель, инициализировать массив путем прямого обращения к каждому его элементу долго и утомительно. Конечно, это обычно делается в цикле, но об организации циклов мы поговорим позднее.

Вот пример общепринятого способа инициализации массива при его объявлении:

```
char word[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

Здесь объявляется безразмерный массив переменных типа `char`, который после заполнения становится строкой `"hello"`. Мы пока не говорим специально о строковых переменных, просто обратите внимание на символ `'\0'`. Это специальный символ конца строки. Правомерной в этом случае будет и такая форма инициализации: `char word[] = "hello";` в этом случае строка автоматически включает в себя нулевой байт конца строки и такое значение справа называется строковым литералом. Как видно из этого примера, при инициализации допускается не указывать количество элементов в объявлении массива, тогда компилятор автоматически определяет количество элементов в массиве по числу значений в списке его инициализации (в вышеприведенном примере элементов будет шесть). Если же объявить последний массив так:

```
char word[10] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

то ничего страшного, в принципе, не произойдет. Компилятор выделит под такой массив десять байт, но старшие четыре байта массива останутся не проинициализированы. Заметим: программисту следует ожидать, что там будет какой-то "мусор", то есть случайные значения, оставшиеся в памяти от каких-то предыдущих данных, а отнюдь не нулевые значения.

Многомерные массивы.

Ничто не мешает объявить массив так: `тип имя [индекс_1] [индекс_2] ... [индекс_n];` например: `int m[4][3];`

Такой массив является массивом массивов - массивом, каждый из элементов которого является, в свою очередь, массивом. Проще говоря, в приведенном примере создастся матрица, имеющая четыре строки и три столбца. Двумерным он называется потому, что управляется при помощи двух индексов (индекс строки, первый при объявлении, и индекс столбца). Существует два варианта инициализации многомерного массива:

```
int m[4][3] = {{11,  15,  30},
               {10,  20,  31},
```

```
                { 5,   8,   0},
                {17,  25,  47}};
```

что полностью эквивалентно варианту:

```
int m[4][3] = {11,  15,  30,
               10,  20,  31,
               5,   8,   0,
               17,  25,  47};
```

поскольку инициализируются все элементы многомерного массива. Если же инициализируются не все элементы, то внутренние фигурные скобки обязательны:

```
int m[4][3] = {{11,  15,  30},
               {10,  -2},
               {5},
               {}};
```

при такой записи первый внутренний массив проинициализируется полностью, во втором останется в неопределенном состоянии последний элемент, в третьем подмассиве - два последних элемента останутся без инициализации, хотя память под них будет выделена, как и под четвертый подмассив. В такой записи:

```
int m[4][3] = {11,  15,  30,
               10,  -2,
               5};
```

компилятор также не найдёт ошибок, хотя результат будет совсем иным, чем в предыдущей записи, а именно: пятерка проинициализирует третий элемент второго подмассива, а третий и четвертый подмассивы останутся без инициализации.

И вкратце о строковых массивах. Запись:

```
word = "goodbye"; // Ошибка! на этапе компиляции
```

Ошибка произойдёт из-за того, что выделенная под массив `word` память может отличаться отличаться объёмом от памяти, которая должна быть задействована для хранения значения `goodbye`. То есть мы не можем записывать литералы в уже существующие массивы. Для выполнения подобных операций в процессе выполнения программы служат функции типа `strcpy`. Но об этом поговорим далее, в разделе Функции.

Тип данных структуры

Дальнейшим развитием идеи объединить под одним заголовком много данных, логически связанных между собой, стало введение в язык С понятия структуры. Фундаментальным отличием структуры от массива является то, что в структуре можно хранить данные разных типов, например фамилию, имя, возраст программиста, и его оклад. Определение структуры выглядит так:

```
struct имя_структуры {
```

```
    объявление_1;
    объявление_2;
    ...
    объявление_n;
};
```

Компоненты структуры называются ее членами (кому не нравится это слово - полями). Объявление членов структуры выглядит так же, как и объявление любой переменной того же типа, рассмотренное ранее. `Имя_структуры` назначается в соответствии с теми же правилами, что и имя любой переменной.

Обращаем внимание, что объявление структуры, приведенное выше, просто создает новый тип данных, но еще не выделяет под эти данные память. После объявления структуры (не обязательно сразу), надо объявить имена данных, которые будут имеют тип этой структуры. И тогда, под эти данные, компилятор выделит память. Допускается объединение в одной конструкции объявления структуры и данных типа этой структуры, а также инициализация памяти, выделенной под данные типа структуры.

```
struct person {
    char name[10];
    char surname[10];
    int age;
    float salary;
} president; // Только после появления в этой строке слова president
             // компилятор реально выделит память под переменную
             // с именем president и полями структуры person

struct person {
    char name[10];
    char surname[10];
    int age;
    float salary;
} president = {"Joe", "Byden", 77, 21000.00}; // Тот же пример с
             // инициализацией по результатам прошедших выборов.
             // Через год в программе потребуется выполнить вот такую команду:

president.age = 78; // Это пример обращения по индексу к членам структуры
president.age += 1; // Или, например, так

// С учетом того, что предстоят еще судебные разбирательства, структуру
следует инициализировать следующим образом:
struct person {
    char name[10];
    char surname[10];
    int age;
    float salary;
} president_1={"Donald", "Trump", 73, 21000.00},// обратите внимание на запятую
  president_2={"Joe", "Byden", 77, 21000.00}; // разделитель между
                                    // president_1 = {...}, president_2 =
{...};
```

Также обращаем внимание, что если структура в качестве своего члена содержит массивы (`char name[10]; char surname[10];` в нашем примере), то размеры этих массивов должны быть указаны явно.

Массивы структур.

А если бы на выборах кандидатов в президенты было больше? Тогда можно было бы записать:

```
struct person {
    char name[10];
    char surname[10];
    int age;
    float salary;
} presidents[10]; // Таким образом память выделяется на 10 кандидатов.
                  // К члену такого массива по индексу следует обращаться так:
presidents[индекс].имя_члена;
```

К структурам также часто применяется оператор `typedef` значительно сокращающий синтаксис определения структуры, а именно

```
typedef struct person {
 char name[10];
 char surname[10];
 int age;
 float salary;
} President;
```

с дальнейшим обращением уже к новому типу, например, для создания массива, `President[10];` В таком описании важно помнить, что при описании структуры память не выделяется, а совершается только определение. поскольку в операторе `typedef` указано и название структуры `struct person` и новое название `President`, объявления `President Pres[10]` и `struct person Pres[10]` эквивалентны по смыслу, а уже после одного из таких объявлений можно обращаться к `Pres[i]`.

Объединения (union)

В языке C++ существует еще один вариант структуры, называемый объединение (`union`).
Используется такой вариант структуры в тех случаях, когда в одной и той же области памяти необходимо хранить данные разных типов. Например, если необходима единственная переменная с именем `x`, которая может содержать и единственный символ, и вещественное число, и целое число, но не одновременно, как структура, а один тип взамен другого.
Различие между структурами и объединениями заключается в механизме выделения памяти под переменные. Так, в нашем примере ниже, переменная x может хранить данные типа `int`, `char`, `float`, но только один тип одновременно. Размер памяти, отводимой компилятором для хранения `union`, равен размеру памяти, необходимой для хранения наибольшего члена (в нашем примере `float`).

Определение объединения выглядит так:

```
union имя_объединения {
 объявление_1;
 объявление_2;
 ...
 объявление_n;
};
```

Определение данных выглядит как `union имя_объединения имя1, имя2, …;` Например можно работать с объединениями так:

```
union example_U {
    int i;
    char c;
    float f;
};
union example_U x;
x.i = 5; // Запись в x целого числа 5
x.f = 3.62; // Запись в x вещественного числа 3.62. Прошлая запись
перезаписана.
```

Инициализация `union` при объявлении раньше допускалась при условии, что может быть инициализирован только первый член объединения указанием константы соответствующего типа, заключенной в фигурные скобки. Т.е. в нашем примере:

```
union example_U {
    int i;
    char c;
    float f;
} x = {5};
 // А вот так для нашего примера было бы недопустимо в старых версиях C:
 union example_U {
 int i;
 char c;
 float f;
} x = {'a'};
```

Чтобы стал допустим второй вариант инициализации, надо предпоследнюю строку в объявлении union переставить на первое место, выше строки `int i;` В C++ этот недочёт был исправлен начиная с самой первой версии, поэтому инициализировать объединения можно как угодно.
Доступ к члену объединения осуществляется, как мы видели в первом примере, с помощью операции `.` (точка). Записав в `union` один тип, получить его из `union` операцией присвоения можно как другой тип. Но автоматического преобразования типов при использовании `union` в качестве аргументов функции компилятор производить не обязан. Т.е. при вот таком использовании `union` в вышеприведенном примере:

```
x.i = 5;
printf("x.f = %f\n", x.f); // распечатается не 5, а мусор.
```

Структуры внутри структур

В языке Си можно определить структуру, содержащую другие структуры в качестве своих членов. При этом внутренние структуры или определяются при определении внешней структуры, т.е. вот так:

```
struct Day {
    char week_day[10];
    struct Date {
        int day, month, year;
    } date; // Это определение данных типа структуры date
} tomorrow; // А это уже, заодно, и определение данных типа структуры day
```

или должны быть определены ранее, до определения внешней структуры, т.е. вот так:

```
struct Date {
    int day, month, year;
};
struct Day {
    char week_day[10];
    struct Date date;
} tomorrow;
```

Дальнейшее использование структуры структур выглядит так (для нашего примера):

```
 strcpy(tomorrow.week_day, "Tuesday");
 tomorrow.date.day = 26;
 tomorrow.date.month = 11;
 tomorrow.date.year = 2020;
```

Как видим, все манипуляции производятся с помощью уже знакомой нам операции `.` (точка), ничего сложного.

Битовые поля

Чтобы заставить компилятор обрабатывать часть байта, в языке Си предусмотрен специальный тип структур - битовые поля. Это позволяет упаковать информацию даже в часть байта, если целый байт избыточен для данного конкретного случая. Точнее говоря, компилятор выделяет для каждого члена такой структуры целый байт, но при помощи специального для битовых полей оператора : (двоеточие), из выделенного байта используется только заданное программистом количество бит. Смысл в том, чтобы присвоить выделенному количеству бит имя.
Вот полезный пример использования битовых полей:

```
struct t_char {
    char chrtr;
    int is_char : 1; // Предполагается, что под каждое поле выделяется по 1
биту
    int is_upp : 1;
    int is_low : 1;
    int is_dig : 1;
    int is_spc : 1;
```

```
};
```

На самом деле под всю структуру выделится четыре байта, один из которых займёт первый параметр `char`, а в остальных распределятся биты. Но да, действительно, несмотря на то, что мы указали пять параметров типа `int` - будет задействовано не двадцать байт. Возможно, такое поведение связано с оптимизациями современных компиляторов, а используя компиляторы для “железа” будут действительно выделяться гораздо меньшие объёмы.

```
struct t_char exdata;
exdata.chrtr = 'w';
exdata.is_char = 1;  // Флаг того, что член экземпляра exdata - символ
exdata.is_upp = 0;
exdata.is_low = 1;  // Флаг того, что chrtr структуры t_char - не заглавный
exdata.is_dig = 0;
exdata.is_spc = 0;
```

В этом примере в одной структуре объединены символ и несколько однобитовых флагов, характеризующих этот символ. Аналогичным образом в такой структуре можно объединить `union` и флаги, характеризующие членом какого типа в текущий момент представлен этот `union`.

Практическое задание

1. Создать и инициализировать переменные пройденных типов данных
2. Создать перечисление с возможными вариантами символов для игры в крестики-нолики
3. Создать массив, способный содержать значения такого перечисления и инициализировать его.
4. * Создать структуру данных «Поле для игры в крестики-нолики» и снабдить его всеми необходимыми свойствами
5. ** Создать объединение и структуру с битовыми флагами указывающими какого типа значение в данный момент содержится в объединении

Дополнительные материалы

Используемые источники

1. *Брайан Керниган, Деннис Ритчи.* Язык программирования С. — Москва: Вильямс, 2015. — 304 с. — ISBN 978-5-8459-1975-5.
2. *Stroustrup, Bjarne* The C++ Programming Language (Fourth Edition)