Конспект для подготовки к экзамену по предмету «Основы программирования»

ВНИМАНИЕ! Возможны опечатки

Преподаватель: **Клименко Виктор Владимирович**

1й семестр, факультет ИТиП, кафедра ИС

Подготовил и оформил: Махонин Кирилл

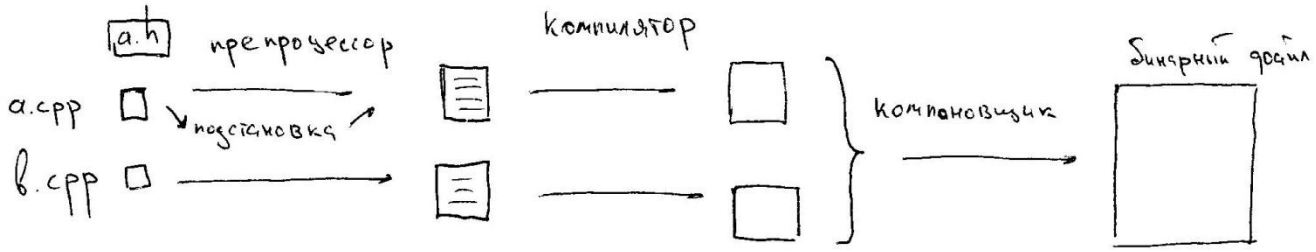
Предложения и ошибки просьба отправлять на 172432@niuitmo.ru

СОДЕРЖАНИЕ

Процесс создания выполняемого кода (препроцессор, компилятор, компоновщик). Сборка проекта, состоящего из нескольких файлов. Показать каким образом и в каком порядке происходит сборка проекта

Препроцессор С — программный инструмент, изменяющий код программы для последующей компиляции и сборки, используемый в языках программирования Си. Этот препроцессор обеспечивает использование стандартного набора возможностей:

- Замена комментариев пустыми строками
- Включение файла — #include
- Макроподстановки — #define
- Условная компиляция — #if, #ifdef, #elif, #else, #endif



Компилятор — программа или техническое средство, выполняющее компиляцию.

Компиляция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному. Входной информацией для компилятора является описание программа на языке С, а на выходе компилятора — эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код).

*Компоновщик (*он же *линкер, редактор связей)* — программа, которая производит компоновку: принимает на вход один или несколько объектных модулей и собирает по ним исполнимый модуль.

Для связывания модулей компоновщик использует таблицы имён, созданные компилятором в каждом из объектных модулей. Такие имена могут быть двух типов:

- Определённые или экспортируемые имена — функции и переменные, определённые в данном модуле и предоставляемые для использования другим модулям;
- Неопределённые или импортируемые имена — функции и переменные, на которые ссылается модуль, но не определяет их внутри себя.

Работа компоновщика заключается в том, чтобы в каждом модуле определить и связать ссылки на неопределённые имена. Для каждого импортируемого имени находится его определение в других модулях, упоминание имени заменяется на его адрес.

Процесс сборки:

- Прохождение препроцессором каждого файла.
- Прохождение компилятором каждого файла (получаем объектный файл, описывающий все функции, переменные и объекты, содержащиеся в коде).

- Объединение **всех файлов проекта**, используя компоновщик.

При динамической линковке в код встраивается вызов функции (если библиотека, необходимая для запуска функции присутствует в операционной системе), а при статической линковке в код программы встраивается код функции.

Встроенные типы данных. Представление данных в памяти. Дополнительный код. Числа с плавающей точкой. Пояснить, каким образом физически могут храниться числа в памяти компьютера

В языке С существуют всего несколько базовых типов данных

- char – один байт содержащий один символ из локального символьного набора
- int – целое число, обычно имеющее типовой размер для целых чисел в данной системе
- float – вещественное число одинарной точности с плавающей точкой
- double – вещественное число двойной точности с плавающей точкой

Кроме того, к целым числам могут применяться модификаторы `short` и `long`. (Слово `int` можно опустить)

```
short int a;
long int b;
long h;
```

Модификатор `signed` (пер.: «со знаком»), использующийся по умолчанию или `unsigned` (пер.: «без знака») может применяться к типу `char` или любому целочисленному. Числа типа `unsigned` всегда неотрицательны. Например, если `char` имеет длину 8 бит, то диапазон значений `unsigned char` [0;255], `signed char` [-128;127].

Данные хранятся в двоичном коде (1 или 0). Это объясняется тем, что электронные элементы, из которых строится оперативная память, могут находиться в одном из двух устойчивых состояний, которые можно интерпретировать как 0 или 1.

Количество информации, которое можно поместить в один элемент памяти называется *битом*. Эти элементы можно объединять и группировать. Последовательность битов, рассматриваемых аппаратной частью как единое целое, называется машинным словом.

Дополнительный код

Дополнительный код двоичного числа получается путем применения к каждому биту числа, кроме значащего (старшего), операцию отрицания (0 -> 1; 1 -> 0) и прибавлением единицы.

Так же можно использовать вычитание данного числа из нуля.

Дополнительный код позволяет заменить операцию вычитания на операцию сложения и тем самым сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел. Для знакового типа старший бит является знаковым (0 – положительное число, 1 - отрицательное).

Для положительных знаковых и беззнаковых чисел:

В двоичном представлении прямой, обратный и дополнительный коды совпадают.

Для отрицательных:

Старший бит - знаковый, остальные биты инвертируются, и к самому младшему биту прибавляется единица.

Пример:

127_{10}: Прямой код = 01111111_2 = Обратный = Дополнительный.

-127_{10}: Прямой код = 11111111_2, Обратный = 10000000_2, Дополнительный = 10000001_2.

Числа с плавающей точкой

Для дробных двоичных чисел можно применять *однобайтное представление*: [знак][порядок][мантисса]. Под знак отводится 1 бит, под порядок 3, под мантиссу 4.

Знак

- 0 – Число положительно
- 1 – Число отрицательно

Порядок показывает, насколько нужно сдвинуть точку относительно мантиссы. Порядок записан в виде нотации с избытком.

Мантисса – значащие цифры числа в прямом представлении.

В двухбайтном представлении под порядок отводится 4 бита, под мантиссу 11.

Пример:

```
1 0101 01101101101
```

- -3 0.000***01101101101***

Ответ: $-\frac{877}{16384}$

Если порядок положителен, то мантисса должна начинаться с единицы. Если отрицателен, то заканчиваться на единицу.

Запоминающие устройства

В основе работы запоминающего устройства может лежать любой физический эффект, обеспечивающий приведение системы к двум или более устойчивым состояниям. В современной компьютерной технике часто используются физические свойства полупроводников, когда прохождение тока через полупроводник или его отсутствие трактуются как наличие логических сигналов 0 или 1. Устойчивые состояния, определяемые направлением намагниченности, позволяют использовать для хранения данных разнообразные магнитные материалы. Наличие или отсутствие заряда в конденсаторе также может быть положено в основу системы хранения. Отражение или рассеяние света от поверхности CD, DVD или Blu-ray-диска также позволяет хранить информацию.

Наиболее известные

- SSD - компьютерное немеханическое запоминающее устройство на основе микросхем памяти.
- HDD - запоминающее устройство, основанное на принципе магнитной записи.

Идентификаторы. Объявление. Инициализация. Пояснить, что такое идентификатор, каков синтаксис объявления. Область видимости переменных

Идентификатор – последовательность букв (к ним также относится символ «_») и цифр. Первый символ обязательно является буквой. Буквы верхнего и нижнего регистра различаются. На идентификаторы могут накладываться ограничения по длине, но это зависит от типа идентификатора и конкретной реализации языка.

В языке С все переменные необходимо объявить до их использования. Обычно это делается в начале функции до любых выполняемых операторов. Объявление состоит из наименования типа и списка переменных, например:

```
int a,b;
char c;
```

Переменные можно *инициализировать* прямо в объявлениях. Если после имени поставить знак равенства и выражение, то значение этого выражения будет присвоено переменной при её создании:

```
int f=4;
```

Внешние (переменные, объявление вне функций) и *статические* переменные по умолчанию **инициализируются нулями**. Автоматические переменные (локальные переменные), для которых инициализирующие выражения не указаны явно, **содержат случайные значения**, «мусор».

Областью видимости называется часть программы, в пределах которой можно использовать имя. Для автоматической переменной, объявляемой в начале функции, областью видимости является эта функция. Локальные переменные с одинаковыми именами, объявленные в разных функциях, **не имеют никакого отношения друг к другу**. То же самое справедливо и для параметров функций, которые, по сути, являются локальными переменными.

Область действия внешней переменной распространяется от точки, в которой она объявлена, до конца компилируемого файла.

Если необходимо обратиться к внешней переменной до её определения или если она определена в другом файле исходного кода, то обязательно нужно вставить объявление с ключевым словом *extern*.

Важно, что объявление, в отличие от определения, сообщает, что переменная обладает определенными свойствами, а определение выделяет место в памяти для её хранения.

Оформление кода, форматирование. Рассказать об общепринятых принципах форматирования. Пояснить на примерах к каким ошибкам может приводить неправильное форматирование

Стандарт оформления кода — набор правил и соглашений, используемых при написании исходного кода. Наличие общего стиля программирования облегчает понимание и поддержание исходного кода, написанного более чем одним программистом, а также упрощает взаимодействие нескольких человек при разработке программного обеспечения.

Обычно, стандарт оформления кода описывает:

- способы выбора названий и используемый регистр символов для имён переменных и других идентификаторов:
  - запись типа переменной в её идентификаторе (венгерская нотация)
  - регистр символов (нижний, верхний, «верблюжий», «верблюжий» с малой буквы)
  - использование знаков подчёркивания для разделения слов;
- стиль отступов при оформлении логических блоков — используются ли символы табуляции, ширина отступа;
- способ расстановки скобок, ограничивающих логические блоки;
- использование пробелов при оформлении логических и арифметических выражений;
- стиль комментариев и использование документирующих комментариев.

Вне стандарта подразумевается ограничение размера кода по горизонтали (чтобы помещался на экране), а также функции или метода в размер одного экрана.

Венгерская нотация

Суть венгерской нотации сводится к тому, что имена идентификаторов предваряются заранее оговорёнными префиксами, состоящими из одного или нескольких символов. При этом, как правило, ни само наличие префиксов, ни их написание не являются требованием языков программирования, и у каждого программиста (или коллектива программистов) они могут быть своими.

Пример

Префикс	Сокращение от	Смысл	Пример
n, i	int	целочисленная переменная	`nSize, iSize`
l	long	длинное целое	`lAmount`
a	array	массив	`aDimensions`

Пример ошибки, вызванной неправильным форматированием (при чтении кода визуально кажется, что `res=0`, когда `a!=1488`, однако `res=0` тогда, когда `temp>=200)`:

```
int temp = 228, a = 1488, res;
if (temp < 200) {
           if ( a == 1488) {
                 res = 1;
           }} else res = 0;
```

Расположение данных в памяти. Доступ к данным, адреса, указатели. Операции с указателями. Рассказать каким образом располагаются данные в памяти, что такое адрес в памяти, что такое указатель, какие операции возможны над указателями

С точки зрения разработчика программного обеспечения оперативная память представляет собой упорядоченную последовательность ячеек — байт, предназначенных для размещения данных, которыми оперирует программа во время своего выполнения.

Упорядоченность означает, что каждый элемент последовательности (каждая ячейка памяти) имеет свой порядковый номер. Этот порядковый номер называют *адресом ячейки памяти* — адресом байта. Непрерывный диапазон ячеек, доступный для адресации в конкретной операционной системе, называют *адресным пространством*.

Указатель – это переменная, в которой хранится адрес другого объекта (как правило, другой переменной). Например, если одна переменная содержит адрес другой переменной, говорят, что первая переменная ссылается на вторую.

Переменная, хранящая адрес ячейки памяти, должна быть объявлена как указатель. Объявление указателя:

```
<тип указателя> *<имя указателя>;
```

Типом указателя служит тип объекта, на который может указывать указатель. С формальной точки зрения указатель любого типа может ссылаться на любое место в памяти. Однако операции адресной арифметики тесно связаны с базовым типом указателей, поэтому важно их правильно объявить.

Оператор получения адреса

Оператор `&` является унарным и возвращает адрес своего операнда.

```
int a=4;          //Численная переменная a, имеющая значение 4
int *b=&a;        //Присвоить указателю b адрес переменной a
```

Адрес и значение переменной никак не связаны друг с другом.

Оператор разыменования

Оператор разыменования `*` является антиподом оператора `&`. Этот унарный оператор возвращает значение, хранящееся по указанному адресу.

```
int a = 4;            //Численная переменная a, имеющая значение 4
int *b = &a;          //Присвоить указателю b адрес переменной a
a = 7;                //Устанавливаем переменной a значение, равное 7
*b = 16;              /* Устанавливаем данным, расположенным по адресу a (b
указывает на a) значение, равное 16 */
printf("%d", a);      //Выведет 16
```

Указатель на функцию

Несмотря на то, что функция не является переменной, она располагается в памяти, и, следовательно, её адрес можно присвоить указателю. Этот адрес считается точкой входа в функцию. Поскольку указатель может

ссылаться на функцию, её можно вызывать с помощью этого указателя. Это позволяет также передавать функции другим функциям в качестве аргументов.

Пример:

```
//Объявим функцию, суммирующую две переменной
int summ(int a, int b){
	return a+b;
}
/*
Объявим функцию, принимающую две переменной и ссылку на функцию рассчета
*/
int calculate(int x, int y, int (*funct)(int a, int b)){
	return (*funct)(x, y); //Возвращаем результат функции funct, вызванной
//с параметрами x и y
}
int main(){
	int c = calculate(5, 4, &summ); //Передаем в функцию два числа и ссылку
//на функцию
	printf("%d", c);//Выводим результат
	return 0;
}
```

Адресная арифметика

К указателям можно применять только две арифметические операции: сложение и вычитание. При увеличении – указатель ссылается на ячейку, в которой хранится следующий элемент базового типа. При уменьшении – он ссылается на предыдущий элемент. (При прибавлении к указателю переменной, имеющей адрес 2000 и размер 2 байта, мы получим 2002, а не 2001). Существет возможность вычитать указатели друг из друга. Так же присутствует возможность сравнения указателей. Для типа *void ** нельзя использовать адресую арифметику, поскольку его размер неизвестен.

Указатель можно присваивать другому указателю:

```
int a = 4;			//Численная переменная a, имеющая значение 4
int *p1, *p2;			//Два указателя на int
p1 = &a;			//Записываем в p1 адрес a
p2 = p1;			//Записываем в p2 то же, что и в p1 (адрес a)
printf("%d", *p2);		//Выведет 4
```

Строки в Си. Операции со строками стандартной библиотеки. Особенности хранения строк в Си. Основные операции со строкам с помощью стандартной библиотеки

В языке С существуют строковые константы. Строка – последовательность символов, заключенная в двойные кавычки. В языке С предусмотрены управляющие символьные константы, которые также называются эскейп-последовательностями.

Знак	**Escape-последовательности**
Новая строка	\n
Горизонтальная табуляция	\t
Backspace	\b
Возврат каретки	\r
Звуковой сигнал	\a
Обратная косая черта	\\
Вопросительный знак	\?
Одиночные кавычки	\'
Двойные кавычки	\"
Нуль-символ	\0

Для представления строк символов используется одномерный массив, последним элементом которого является нулевой байт (он же признак завершения строки). Это единственный вид строки, предусмотренный в языке С.

Объявляя массив символов, следует зарезервировать одну ячейку для нулевого байта, т.е. указать размер, на единицу больше, чем длина наибольшей предполагаемой строки. Например, чтобы объявить массив str, предназначенный для хранения строки, состоящей из 10 символов, следует выполнить следующий оператор.

char str[11];

В этом объявлении предусмотрена ячейка, в которой будет записан нулевой байт. К строковым константам компилятор автоматически добавляет нулевой байт.

strcpy (d, s) – Копирует s в d
strncpy(d, s, n) – Копирует не более n символов из s в d
strcat(d, s) – Присоединяет s в хвост к d
strncat(d, s, n) – Присоединяет n символов из s в хвост к d
strcmp(a, b) – Дает отрицательное, нуль или положительное число при a < b, a == b, a > b соответственно
strncmp(a, b, n) – То же самое, что и strcmp(a,b), но только для первых n символов

strchr(s, f) – Возвращает указатель на первый символ f в s или NULL, если символ не найден
strrchr(s, f) – Возвращает указатель на последний символ f в s или NULL, если символ не найден
strlen(s) – Вычисляет длину s

Пример реализации функций:

```
char * strcpy(char * dest, const char * source){
	char * start = dest;
	while (*source){
		*dest=*source;
		dest++;
		source++;
	}
	*dest=0;
	return start;
}
char * strncpy(char * dest, const char * source, int n){
	char * start = dest;
	while ((*source)&&(n>0)){
		*dest=*source;
		dest++;
		source++;
		n--;
	}
	if (n>0)
		*dest=0;
	return start;
}
char * strcat(char * dest, const char * source){
	char * start = dest;
	while (*dest)
		dest++;
	while (*source){
		*dest=*source;
		dest++;
		source++;
	}
	*dest=0;
	return start;
}
char * strncat(char * dest, const char * source, int n){
	char * start = dest;
	while (*dest)
		dest++;
	while ((*source)&&(n>0)){
		*dest=*source;
		++dest;
		++source;
		--n;
	}
	if (n>0)
		*dest=0;
	return start;
}
```

```
int strcmp(const char * a, const char * b){
	while (*a==*b){
		if (*a==0)
			return 0;
		++a;
		++b;
	}
	if (*a<*b)
		return -1;
	return 1;
}
int strncmp(const char * a, const char * b, int n){
	while (*a==*b){
		if ((*a==0)||(n<=1))
			return 0;
		++a;
		++b;
		--n;
	}
	if (*a<*b)
		return -1;
	return 1;
}
char * strchr(char * str, char x){
	while (*str){
		if (*str==x)
			return str;
		++str;
	}
	return 0;
}
char * strrchr(char * str, char x){
	char * last = 0;
	while (*str){
		if (*str==x)
			last = str;
	}
	return last;
}
int strlen(char * x){
	char * start = x;
	while (*x)
		++x;
	return x-start;
}
```

Операции. Типы операций, приоритеты

Операция – конструкция в языках программирования, аналогичная по записи математическим операциям, то есть специальный способ записи некоторых действий.

Операнд – аргумент операции.

Операторы, имеющие одинаковый приоритет, выполняются слева направо. Разумеется, для изменения порядка вычисления операторов можно применять скобки. Скобки позволяют присвоить некоторым операторам или группе операторов более высокий приоритет.

Типы операций

- Унарная – Один аргумент (`i--`)
- Бинарная – Два аргумента (`a+b`)
- Тернарная – Три аргумента (`f?a:b`)

Синтаксис операций

- Префиксная нотация (`++a`)
- Инфиксная нотация (`a+b`)
- Постфиксная нотация (`b--`)

`.`	прямой выбор	Постфиксный
`->`	опосредованный выбор	Постфиксный
`++ --`	положительное и отрицательное приращение	Постфиксный
`++ --`	положительное и отрицательное приращение	Префиксный
`~`	побитовое НЕ	Унарный
`!`	логическое НЕ	Унарный
`- +`	изменение знака, плюс	Унарный
`&`	адрес	Унарный
`*`	опосредование (разыменование)	Унарный
`* / %`	мультипликативные операции	Бинарный
`+ -`	аддитивные операции	Бинарный
`<< >>`	сдвиг влево и вправо	Бинарный
`< > <= >=`	отношения	Бинарный
`== !=`	равенство/неравенство	Бинарный
`& ^ |`	побитовое И, исключающее ИЛИ, ИЛИ	Бинарный
`&& ||`	логическое И, ИЛИ	Бинарный
`? :`	условие	Тернарный
`= += -= *= /= %= <<= >>= &= ^= |=`	присваивание	Бинарный

Важно, что `a++` возвращает исходное значение перед его изменением, в то время, когда `++a` возвращает измененное значение.

Side effect (изменение значений) и result (результат)

	side effect	result
a+b	-	сумма a и b
a++	a увеличивается на 1	старое значение a
++a	a увеличивается на 1	новое значение a
a=b	присвоение a значения b	b
!a	-	!a
(type)a	-	(type)a

Lvalue и rvalue

```
<lvalue>=<rvalue>;
```

Для того, чтобы отличать выражения, обозначающие объекты, от выражений, обозначающих только значения, ввели понятия lvalue и rvalue. Слово lvalue используется для обозначения выражений, которые могут стоять слева от знака присваивания (left-value). Им противопоставлялись выражения, которые могли находиться только справа от знака присваивания (right-value).

Битовые и логические операции

В основе *операторов сравнения*, и *логических операторов* лежат понятия «истина» и «ложь». **В языке С истинным считается любое значение не равное нулю.** Ложное значение всегда равно 0. Выражения, использующие операторы сравнения и логические операторы, возвращают 1 и 0, если результат истинен и ложен соответственно.

Язык С содержит полный набор *побитовых операторов*. Поскольку язык С был разработан в качестве замены языка ассемблера, он предусматривает многие операции низкого уровня, в частности, побитовые операции, предназначенные для проверки, установки и сдвига битов, из которых состоят байты и машинные слова, образующие переменные типа `char` и `int`. Побитовые операции нельзя применить к переменным типа `float, double, long double, void` и других, более сложных типов.

Операторы сравнения и логические операторы

Операторы сравнения	
`>`	Больше
`>=`	Больше или равно
`<`	Меньше
`<=`	Меньше или равно
`!=`	Не равно
`==`	Равно
Логические операторы	
`&&`	И
`||`	ИЛИ
`!`	НЕ

Операторы сравнения и логические операторы имеют более низкий приоритет, чем арифметические операторы. Таким образом `9 > 5 + 3` будет вычислено так, будто оно записано, как `9 > (5 + 3)`.

Побитовые операции И, *ИЛИ* и *НЕ (дополнение до единицы)* описываются теми же таблицами истинности, что и их логические аналоги, за исключением того, что теперь все сравнения проводятся поразрядно. Оператор *исключающее ИЛИ* возвращает единицу, если аргументы не равны.

Примеры

Операция	A&B *(И)*	A|B *(ИЛИ)*	~A *(НЕ)*	A^B *(Исключающее ИЛИ)*
A	10101110	00010001	00010010	01011010
B	01100011	01001101	–	11110111
Результат	00100010	01011101	11101101	10101101

Помните, что результатом работы операторов сравнения и логических операторов всегда являются истинные или ложные значения, в то время как аналогичные побитовые операторы могут порождать любые целые числа. Иными словами, результатом побитовых операций является любые числа, а не только 0 или 1, как при использовании логических операторов.

Операторы ветвления. Синтаксис, особенности. Смежные вопросы: понятие итерации, условия, пред- и пост- условия, инварианты циклов

Оператор ветвления (условный оператор) — оператор, обеспечивающий выполнение определённой команды (набора команд) только при условии истинности некоторого логического выражения, либо выполнение одной из нескольких команд (наборов команд) в зависимости от значения некоторого выражения.

В языке предусмотрены два условных оператора: `if` и `switch`. Кроме того, в качестве альтернативы условному оператору `if` можно применять тернарный оператор `?`.

Оператор if

```
if (<выражение>) <оператор1>;
else <оператор2>;
```

`Оператор1` и `оператор2` могут состоять из одного или нескольких операторов или отсутствовать вовсе (пустой оператор). Раздел `else` является необязательным.

Если `выражение` истинно (т.е. не равно 0), выполняется `оператор1`, в противном случае выполняется `оператор2`.

```
int x;                          //Объявляем переменную x
scanf("%d",&x);                 //Запрашиваем её с клавиатуры
if (x==4){                      //Если она равна 4..
     printf("Bingo! ");         //Выводим: "Bingo! "
     printf("You win! ");       //Выводим: "You win! "
}
else                            //Иначе (если переменная не равна 4)
     printf("You lose! ");      //Выводим: " You lose! "
```

Тернарный оператор

Оператор `?` называется тернарным, поскольку имеет три операнда.

```
<выражение1>?<выражение2>:<выражение3>
```

Сначала выполняется `выражение1`. Если оно истинно, вычисляется `выражение2`, и его значение становится значением всего тернарного оператора, иначе вычисляется `выражение3`.

```
int x = 10;                 //Переменная x, имеющая значение 10
int y = x>5 ? x-4 : x+3;    //y=x-4, если x>5, иначе y=x+3
printf("%d", y);            //Выводим 6
```

Оператор switch

В языке предусмотрен оператор многовариантного ветвления switch, который последовательно сравнивает значение выражения со списком целых чисел или символьных констант. Если обнаруживается совпадение, выполняется оператор, связанный с соответствующей константой.

```
switch (<выражение>){
     case <константа1>:
          <последовательность операторов1>
```

```
            break;
      case <константа2>:
            <последовательность операторов2>
            break;
      case <константа2>:
            <последовательность операторов2>
            break;
      default:
            <последовательность операторов по умолчанию>
}
```

Выражение может являться символом или целым числом. Например, выражения, результатом которых является число с плавающей точкой, не допускаются. Значение выражения последовательно сравнивается с константами, указанными в операторах case. Если обнаруживается совпадение, выполняется последовательность операторов, связанных с данным оператором case, пока не встретится оператор break или не будет достигнут конец блока switch. Если значение не совпадает ни с одной из констант, выполняется необязательный оператор default.

В языке наложено ограничение на количество операторов case внутри одного switch – максимум 257 операторов.

```
int x=5;
int y;
switch(x){
case 12:          //Пропускаем этот оператор, поскольку x!=12
      y=20;
      break;
case 5:           //Входим внуть этого оператора, поскольку x==5
      y=23;       //Присвоим y значение 23
case 7:           //Поскольку не было оператора break, выполняем следующий case
      y=90;       //Присвоим y значение 90
      break;      //Выходим из оператора case
case 19:          //До этого оператора не доходим
      y=20;
      break;
default:
      y=21;
};
printf("%d %d",x,y);        //Выведем “5 90”
```

Операторы цикла

В языке С, как и во всех других современных языках программирования, операторы цикла предназначены для выполнения повторяющихся инструкций, пока действует определенное правило. Это условие может быть как задано заранее (оператор for), так и меняться во время выполнения цикла (оператор while, do-while).

```
while (<выражение>) <оператор>
do <оператор> while (<выражение>)
for (<выражение1(необ)>; <выражение2(необ)>; <выражение3(необ)>) <оператор(необ)>
```

В операторах while и do выполнение оператора (тела цикла) повторяется до тех пор, пока выражение не станет равно 0. Выражение должно иметь арифметический тип или быть указателем. В операторе while вычисление выражения и проверка условия выполняется перед каждым выполнением телом цикла, а в

операторе do эта проверка выполняет после. Так же выражение1, выражение2, выражение3 могут состоять из операций, разделенных “ , ” , это выполняет сразу все операции.

```
for (i = 0, j = 0; i<n; i++, j++)     //Инициализируем массив i=0, j=0;
// С каждым шагом увеличиваем i и j на 1
      a[i] = a[j];
```

В операторе for выражение1 вычисляется один раз, тем самым осуществляя инициализацию цикла. Выражение2 должно иметь арифметический тип или тип указателя, оно вычисляется перед каждой итерацией (проходом) цикла. Как только значение становится равным 0, цикл прекращает свою работу. Выражение3 вычисляется после каждой итерации и таким образом выполняет повторную инициализацию цикла.

Любое из трех выражений (выражение1, выражение2 или выражение3) или оператор можно опустить.

Операторы перехода

Оператор continue может находиться только внутри цикла. Он вызывает переход к следующей итерации самого внутреннего цикла, содержащего его.

Оператор break можно использовать только в операторах цикла или switch. Он завершает работу самого внутреннего цикла или switch, содержащего данный оператор break, после чего управление переходит к следующему оператору после только что завершенного тела.

С помощью оператора return функция возвращает управление в программу, откуда была вызвана. Ситуация, когда нормальный ход операций приводит в конец функции (к последней фигурной скобке), равносильна выполнению оператора return без выражения.

Оператор goto позволяет перемещать между частями программы. Для оператора необходима метка, представляющая собой идентификатор с двоеточием. Метка должна находиться в той же функции, что и оператор goto (перепрыгивать из функции в функцию нельзя).

```
int x=10;
loop:                       //Метка для перехода
      printf("Hello!\n");   //Выводим "Hello!"
      x--;                  //Уменшьнаем x
      if (x>0) goto loop;   //Если x>0: переходим на метку loop
```

Программа, приведенная выше, 10 раз выведет строку "Hello!".

Массивы. Объявления, инициализация. Внутреннее устройство массивов, связь с указателями и памятью. Понятие массива с математической точки зрения. Операции над массивами данных в стандартной библиотеке (memcpy). Динамическое выделение памяти (malloc, calloc, free)

Массив – это совокупность переменных, имеющих одинаковый тип и объединенных под одним именем. Доступ к отдельному элементу массива осуществляется с помощью индекса. Согласно правилам языка С, все массивы состоят из смежных ячеек памяти (все элементы массива расположены в памяти последовательно).

Объявлением массива служит конструкция вида

```
<тип элементов> <имя массива>[<количество переменных>];
```

Пример:

```
float a[3]; //Массив из трех элементов типа float: a[0],a[1],a[2]
```

Количество элементов массива (размер массива) должно задаваться константным выражением.

Для обращения к элементам массива используется конструкция `a[i]`, где a – указатель на массив, i – индекс элемента, значение которого мы хотим получить или задать.

Имя массива является указателем на первый его элемент (первый элемент имеет индекс 0). Адрес первого элемента массива можно также вычислить с помощью оператора `&`. Например, выражения `sample`, `&sample[0]` эквивалентны. Однако такая конструкция редко используется.

Также, для обращения к элементам массива можно использовать *адресную арифметику*: адрес определенного элемента массива вычисляется, как сумма указателя на массив (являющегося также адресом первого элемента) и позиции искомого элемента в массиве (индекса).

```
int a[2];  //Объявляем массив двух целочисленных переменных
a[0]=12;   //Первому элементу массива a присваиваем значение 12
*(a+1)=26; //Второму, с помощью адресной арифметики, присваиваем 26
```

В языке С допускается инициализация массивов при их объявлении. Общий вид инициализации массива не отличается от инициализации обычных переменных.

```
<тип элементов> <имя массива>[<количество переменных>]={список значений};
```

Список значений представляет собой список констант, разделенный запятыми. Тип констант должен быть совместимым с типом массива. При инициализации массива списком значений можно не указывать размер массива, тогда он будет равен размеру списка значений.

Если размер массива указан, то при передаче списка значений, имеющего больший размер, чем массив, компилятор будет указывать на ошибку. Однако, при передаче списка значений, меньшего, чем размер массива, оставшиеся элементы будут заполнены нулями.

```
int a[3];              //Объявление без инициализации
int b[3]={1,2,3};      //Массив имеет значения (1,2,3)
int b[3]={1,2,3,3};    //Ошибка: размер списка больше размера массива
int c[3]={1,2};        //Массив имеет значения (1,2,0)
```

Если количество элементов массива становится известно только во время выполнения программы, то массив можно получить с помощью динамического распределения памяти.

Термин динамическое распределение памяти означает, что программа может получать необходимую ей память уже в ходе своего выполнения. В языке С предусмотрена система ля динамического распределения памяти. Рассмотрим её.

Память, выделяемая функциями динамического распределения, находится в куче (heap), которая представляет собой область свободной виртуальной памяти, расположенную между кодом программы, сегментом данных и стеком. Хотя размер кучи заранее не известен, её объем обычно достаточно велик.

В основе системы динамического распределения памяти лежат несколько функций.

`void * malloc ( size_t size )` – выделяет указатель на кусок памяти, размером *size* байт. Возвращает адрес или 0 (при ошибке)

`void * calloc ( size_t num, size_t size )` – выделяет указатель на кусок памяти, достаточный для хранения *num* элементов размера *size*. Возвращает адрес или 0 (при ошибке)

`void free ( void * ptr )` – очищает блок памяти, ассоциированный с указателем *ptr*

`void * memcpy ( void * destination, const void * source, size_t num )` – копирует кусок памяти длиной *num*, начиная с указателя *source* в *destination.* Возвращает *destination*

`void * realloc ( void * ptr, size_t size )` – устанавливает размер куска памяти, размещенного по адресу *ptr*, равный *size*. Возвращает новый адрес или 0 (при ошибке)

Дополнительно: `size_t` – целочисленный тип данных, можно применять `int/short/long`

Расположение в библиотеках:

`malloc, calloc, free, realloc` `<stdlib.h>`

`memcpy` `<string.h>`

При каждом вызове функции `malloc()` или `calloc()` выделяют память, которую, после использования необходимо вернуть операционной системе функцией `free()`.

```
char *p;   //Объявим указатель p
p = (char *)malloc(10 * sizeof(char)); /* Выделим размер памяти, достаточный для того, чтобы поместить в него 10 символов (10* размер элемента типа char). Также нам необходимо привести тип void * к типу char *
*/
p[0] = 'a';      //Присваиваем первому элементу значение a
p[1] = 0;  //Второй элемент – терминальный 0
printf("%s", p);//Выводим строку (поскольку p – массив char'ов).
//Программа выведет: "a"
free(p);   //Освобождение памяти
```

Если аргументом функции является одномерный массив, её формальный параметр можно объявить тремя способами:

1. Указатель: `void funct(int *x){`
2. Массив фиксированного размера: `void funct(int x[10]){`

3. Массив неопределенного размера: `void funct(int x[]){`

Эти три объявления эквивалентны друг другу, поскольку их смысл совершенно одинаков: во всех случаях в функцию передается указатель на целочисленную переменную. Размер массива, передаваемого в функцию (способ №2), не имеет никакого значения из-за отсутствия проверки за выход массива.

В языке С не предусмотрена проверка за выход массива. Выход за границы массива (попытка чтения/записи несуществующего элемента) может привести к непредсказуемым результатам и поведению программы. Это является одной из самых распространенных ошибок.

Двумерные массивы

При написании программы может пригодиться возможность сохранять матрицы, наборы пар значений и прочую информацию, которую можно представить в виде таблиц.

Например, нам необходимо заполнить первую матрицу и сгенерировать вторую:

$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$ и $\begin{bmatrix}0 & 0\\1 & 2\\2 & 4\end{bmatrix}$ (элементы второй матрицы рассчитываются, как $a_{i0} = i, a_{i1} = i * 2$)

Обычный массив является набором значений, двумерный массив является набором указателей на одномерные массивы.

Двумерный массив, как и одномерный, можно реализовать либо с использованием динамической памяти, либо без использования её.

Без использования динамической памяти:

```
<тип элементов> <имя массива>[<N>][<M>];
```

- где $N \times M$ размерность массива.

Пример применения

```
int g[4][5];     //Объявление без инициализации
int a[2][2]={{1,2},{3,4}}; //Объявление с инициализацией
printf("%d %d\n%d %d",a[0][0],a[0][1],a[1][0],a[1][1]);     //Вывод
```

Программа выведет:

```
1 2
3 4
```

С использование динамической памяти:

При использовании динамической памяти нам необходимо последовательно выполнить несколько шагов:

1. Объявить указатель на двумерный массив
2. Присвоить указателю память, необходимую для хранения массива указателей на одномерные массивы
3. Каждому элементу массива указателей на одномерные массивы выделить необходимую ему память

После работы с массивом, его, как и одномерный, необходимо освободить.

1. Освободить каждый элемент массива указателей на одномерные массивы
2. Освободить массив указателей

```
int i;           //Переменная для цикла
int** a;         //Указатель на двумерный массив
a = (int **)malloc(3*sizeof(int*));   /* Выделяем память для массива указателей
*/
for (i=0; i<3; i++){  //Каждому из указателей на одномерные массивы
      a[i]=(int *)malloc(2*sizeof(int));    //Выделяем память
      a[i][0]=i;       //Выполняем простые рассчеты
```

```
      a[i][1]=i*2;     //Еще рассчеты
}
printf("%d%d\n%d%d\n%d%d",a[0][0],a[0][1],a[1][0],a[1][1],a[2][0],a[2][1]);
                        //Выводим массив на экран
for (i=0; i<3; i++)
      free(a[i]);           //Освобождаем каждый одномерный массив
free(a);                    //Освобождаем массив указателей
```

Программа выведет:

```
0 0
1 2
2 4
```

Так же стоит заметить, что массивы, выделяемые каждым вызовом malloc в цикле, не обязаны идти друг за другом в памяти, поэтому применять адресную арифметику в двумерных массивах не рекомендуется.

В целом, поскольку двумерные массивы реализуются с помощью обычных, одномерных массивов, то работа с ними очень похожа на работу с одномерными массивами.

Подпрограммы. Объявления, реализация. Механизмы передачи и возврата данных. Механизм вызова подпрограмм. Синтаксис объявления и реализации функций. Случай объявления и реализации в разных файлах, пояснить работу компилятора и компоновщика

Основным структурным элементом языка является функция. Именно функции являются блоками, из которых строится программа. Функции – это удобный способ свести в одно место некоторые вычислительные операции, а затем обращаться к ним много раз. Если функция написана правильно, нет нужды знать, как она работает, достаточно знать, что именно она делает.

Каждая функция (помимо main) должна иметь свое определение перед реализацией самой функции и теми участками кода, откуда функция вызываются.

Определением функции является конструкция

```
<тип возвращаемого значения> <имя функции>(<объявление параметров>);
```

А реализацией функции является

```
<тип возвращаемого значения> <имя функции>(<объявление параметров>){
      <объявления>
      <операторы>
}
```

Рассмотрим функцию, рассчитывающую значение числа base, возведенного в степень n

Для начала, напишем её определение

```
int power(int base, int n);
```

Далее, мы можем писать её реализацию:

```
int power(int base, int n){
/*
      Блок объявления переменных
*/
      int i;
      long result;
/*
      Блок операторов
*/
      for (i = 0; i < n; i++)
            result *= base;
      return result;        //Возвращение результата
}
```

Стоит заметить, что обязательное присутствие объявления функции и расположение объявления переменных функции в её начале обязательно только для чистого C *(преподаватель про это не упоминал и опускал определение функций и обязательность порядка объявлений переменных в начале функции)* .

Так же важно, что в языке C все аргументы функций передаются «по значению». Это означает, что вызываемая функция получает значения своих аргументов в виде переменных, а не оригиналов,

следовательно, вызываемая функция не может модифицировать переменные в вызывающей функции, она в праве менять только свои локальные, временные копии этих переменных.

Для удобства создания больших программ предусмотрен механизм, позволяющий разбивать код на отдельные файлы.

Рассмотрим пример

Заголовочный файл `summ.h`

```
int summ(int a, int b);
int multiply(int a, int b);
```

Файл `summrel.c`

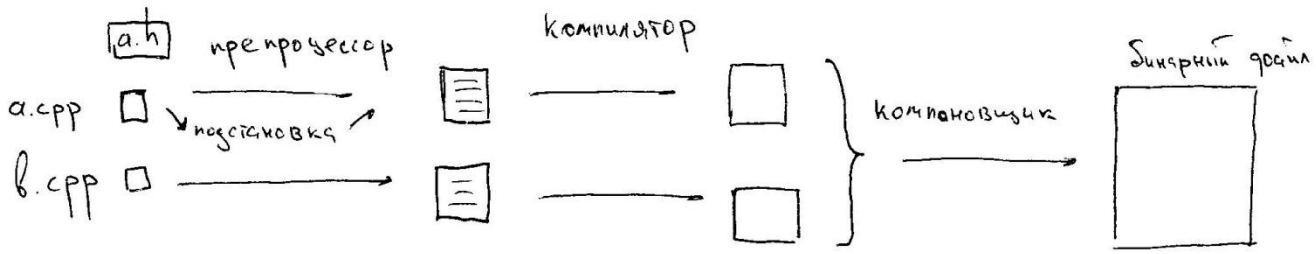
```
#include "summ.h"
int summ(int a, int b){          //Реализуем функцию, описанную в summ.h
      return a+b;
};
int multiply(int a, int b){      //Реализуем функцию, описанную в summ.h
      return a*b;
};
int divide(int a, int b){        //Реализуем функцию
      return a/b;
};
```

Файл `prog.c`

```
#include "summ.h"
#include <stdio.h>

int main(){
      printf("%d %d\n",summ(4,9),multiply(5,10));
      printf("%d\n", divide(80,9)); //Ошибка: функция divide не известна,
//поскольку она не объявлена в summ.h
}
```

Объявление функции `summ` находится в файле `summ.h`, реализация в файле `summrel.c`, а используется функция в `prog.c`.



При запуске сборки препроцессор обрабатывает все файлы по отдельности, затем компилятор компилирует каждый из файлов. По завершении компиляции компоновщик соединяет все файлы в единый бинарный файл. *(Более подробная информация в первом билете)*

Пользовательские типы данных. Перечисляемый тип. Структуры. Псевдонимы для типов. То есть enum, struct, typedef. Отличия typedef от макросов (#define).

В С совсем немного возможностей по построению пользовательских типов данных. Точнее, сами возможности практически безграничны, но вот "базовых кирпичиков", из которых все строится, всего 4. Это структуры (structures), объединения, или союзы (unions), битовые поля (bit fields) и перечисления (enumerations). Причем последний тип, перечисление, в С не очень популярен - никаких дополнительных преимуществ, кроме улучшения читаемости программы, он не дает.

Структура

Структура – это совокупность нескольких переменных, часто различных типов, сгруппированных под одним именем для удобства обращения. Благодаря ним удобно организовывать сложные данные, т.к. их можно рассматривать как единое целое.

Структуры можно копировать, присваивать, передавать в функции и возвращать из функций, однако их нельзя сравнивать. Автоматические структуры также можно инициализировать.

`struct <имя(необ.)> { <элементы> } <список элементов(необ.)>;`

Примеры

```
struct {
      int x;
      int y;
} temp;
temp.x = 12;
```

В приведенном выше варианте мы имеем один экземпляр структуры и не можем создавать прочие экземпляры, так как имя структуры не установлено.

```
struct point {
      int x;
      int y;
};
point temp;
temp.x = 17;
```

В данном варианте мы вводим новый тип данных point. Теперь возможно создавать переменные такого типа данных: `point p1, p2;`

Структуру можно инициализировать, поставив после определения список значений-констант:

```
point temp = { 10, 15 };
```

Расширенными операциями над структурами являются копирование или присваивание структуры как целого, взятие ее адреса операцией `&`, а так же обращение к её элементам.

Обращение к элементам структуры производится с помощью конструкции

`<экземпляр структуры>.<элемент>`

Если у нас есть указатель на экземпляр структуры, то мы можем обращаться к элементам структуры, используя конструкцию

```
<указатель на экземпляр структуры>-><элемент>
```

Так же существует возможность объявлять функции внутри структуры, однако реализация описывается вне функции. Однако такая возможность не предусмотрена стандартом С.

Пример

```
struct square {
      int x;
      int y;
      int a;
      int area();      //Функция
} s1;                  //Создаем тип данных square и переменную s1
int square::area(){    //Реализуем функцию area структуры square
      return a * a;    //Возвращаем результат перемножения полей структуры a
}
void main(){
      square * p = &s1;     //p – указатель на s1
      s1.x = 2;
      s1.y = 10;
      p->a = 40;
      printf("%d", p->area());   //Вызываем функцию данной структуры

}
```

Определение новых типов

Объявление, начинающееся с ключевого слова `typedef`, определяет *новое имя для типа*, а не новую переменную какого-либо существующего типа.

Например, чтобы ввести синоним типа `int` под названием `size`, следует применить конструкцию:

```
typedef int size;
```

Однако, с помощью `typedef` вводятся лишь новые синонимы существующих типов, а не сами новые типы. Следовательно, эти синонимы можно использовать параллельно именам существующих типов.

Объединения

Объединение – структура данных, члены которой расположены по одному и тому же адресу. С помощью объединений можно работать с данными различных типов в пределах одного участка памяти.

Обращение к элементам объединения выполняется так же, как к элементам структур.

Фактически объединение является структурой, в которой все элементы имеют нулевое смещение от ее начала,, сама она имеет достаточную длину, чтобы в нее поместился самый длинный элемент, и при этом выравнивание выполняется правильно для всех типов данных в объединении. Над объединениями разрешено выполнять те же операции, что и над структурами: присваивать или копировать как единое целое, брать адрес и обращаться к отдельным элементам.

Перечисления

Перечисление – тип данных, чьё множество значений представляет собой набор констант. Перечисляемый тип (перечисление) определяется списком **целочисленных** констант, например:

```
enum Boolean { NO, YES };
```

Первая константа в списке получает значение 0, вторая – 1 и т.д., если не указаны явные значения. Если заданы не все значения, те константы, для которых они не указаны, получаю значения, следующие по порядку за последним из указанных.

```
enum Color { WHITE, ORANGE = 3, BLACK };
```

`WHITE` принимает значение 0, `ORANGE` – 3, `BLACK` – 4.

Имена в различных перечислениях должны быть различными. Значения констант в одном перечислении не обязаны быть различными.

```
enum Bool { NO, YES=0 };   //Корректная конструкция, NO = 0, YES = 0
```

Отличие typedef от #define

```
#define ptr int*


typedef pointer int*;
pointer    a, b; // a и b типа int*
ptr        c, d;  // c - int*, d - просто int
```

Препроцессор языка Си. Макросы (с параметрами и без), недостатки использования макросов, альтернатива макросам. Удаление макросов. Условная компиляция. Include guard. Комментарии.

Препроцессор языка Си уже упоминался ранее в первом билете.

```
#define <имя> <текст-для-замены>
```

Это – макроопределение простейшего вида. Всякий раз, когда `имя` встретится в тексте программы после этого определения, оно будет заменено на `текст-для-замены`. Текст для замены может быть произвольным. Обычно текст для замены умещается на одной строке, но если он слишком длинный, его можно разбить на несколько строк с использованием символа продолжения `\`.

Область действия имени, заданного в директиве #define действует до конца файла исходного кода или до директивы `#undef`, которая отменяет определение.

```
#define TAX 0.3
#undef TAX
#define TAX 0.5
…
int price = price + price * TAX;
…
```

Можно также определять макросы с аргументами

```
#define max(A, B) ((A) > (B) ?  (A) : (B))
```

Хотя внешне он напоминает вызов функции, на самом деле max разворачивается прямо в тексте программы путем подстановки. Вместо формальных параметров (A и B) будут подставлены фактические аргументы, обнаруженные в тексте. И при вызове макроса

```
x = max(a+b, c+d);
```

Код при компиляции будет заменен на

```
x = ((a+b) > (c+d) ? (a+b) : (c+d));
```

Так же существует возможность управлять самой препроцессорной обработкой, используя условные директивы, которые выполняются по ходу этой обработки. Это делает возможным условное включение фрагментов кода в программу в зависимости от условий, выполняющихся в момент компиляции.

В директиве `#if` выполняется анализ целочисленного выражения. Если выражение не равно нулю, в программу включаются все последующие строки вплоть до директивы `#endif`, `#elif` или `#else`. (Директива `#elif` аналогична `else if`).

Директивы #ifdef и #ifndef – это специальные формы условной директивы для проверки того, определено то или иное имя или нет.

```
#ifndef TAX
#define TAX 0.5
#endif
```

Работа с файлами. Режимы открытия файлов. Чтение и запись в текстовом (форматированном) и в бинарном режимах. То есть fopen, fclose, fprintf, fscanf, fread, fwrite. Общие сведения о потоках ввода-вывода в языке Си.

Перед тем как читать из файла или записывать в файл, его необходимо открыть с помощью функции `fopen` с определенным режимом открытия (чтение, запись).

После открытия, мы можем рассматривать работу с файлом, как работу с трубой, в которой расположены шары с данными. Мы можем взять шар из трубы, прочитать что на нем написано и положить обратно или же не класть обратно. Так же туда можно помещать свои шары с данными и смотреть, остались ли там еще шары.

После работы с файлом необходимо его закрыть, используя функцию `fclose`. Если файл не будет зарыт, то есть вероятность потери данных.

Запись и чтение производится с помощью функций `fprintf` и `fscanf`, которые являются потоковыми аналогами функций `printf` и `scanf`.

Пример

```
#include <stdio.h>
void main(){
	FILE *f = fopen("data.txt","w");//Открываем файл на запись
	fprintf(f, "This is text for file %d",10); //Записываем в файл
	fclose(f); //Закрываем файл
}
```

Так же возможно использовать функции бинарного чтения и записи (`fread` и `frwrite` соответственно) для чтения и записи определенных **участков памяти**.

Пример

```
#include <stdio.h>

struct _data {
	int j;
	int f;
	long h;
} temp;
void main(){
	FILE *f = fopen("data.bin","r");//Открываем поток для работы с файлом
	fread(&temp,sizeof(_data),1,f);		//Читаем данные в бинарном режиме
	printf("%d %d %ld\n",temp.j,temp.f,temp.h);//Выводим полученные данные
	fclose(f);				//Закрываем файл
}
```

Такой же принцип работы с данными распространяется на:

- Файлы
- Консоль
- Сетевой socket
- Драйвер

Понятие спецификации языка.

Спецификацией языка может являться

- Наличие проверки границ массивов в некоторых компиляторах
- Результат выполнения операций типа: `++I + ++I; I++ + ++I`
- Ограничение длины идентификатора
- Ограничение количества цифр числа по умолчанию в printf

Явное и неявное преобразование типов

В С различают явное и неявное преобразование типов данных. Неявное преобразование типов данных выполняет компилятор С, ну а явное преобразование данных выполняет сам программист. О преобразовании типов данных скажу следующее: «Результат любого вычисления будет преобразовываться к наиболее точному типу данных, из тех типов данных, которые участвуют в вычислении». Для наглядного примера представлю таблицу с преобразованиями типов данных. В таблице рассмотрим операцию деления. В качестве целочисленного типа данных возьмем int, ну и вещественный тип данных у нас будет float.

x	y	Результат деления	Пример
делимое	делитель	частное	x = 15 y = 2
int	int	int	15/2=7
int	float	float	15/2=7.5
float	int	float	15/2=7.5

Из таблицы видно, что меняя переменные различных типов данных местами, результат остается тот же (в нашем случае это делимое и делитель). О неявном преобразовании типов данных все, что же касается явного преобразования, то оно необходимо для того чтобы выполнять некоторые манипуляции, тем самым меняя результата вычисления. Самый простой способ явного преобразования типов данных, пример: допустим нам необходимо разделить такие числа `15` и `2`. Результатом `15/2` является `7`, результатом же `15.0/2` является `7.5` (при таком делении число `15` является вещественным, значит и результат будет вещественный). Само число `15` с точки зрения математики не изменилось, ведь `15=15.0`. Этот же прием можно было применить к двойке, результат был бы тем же, а можно было сразу к двум числам.

Помимо этого можно явно преобразовывать тип используя конструкцию

```
(<новый тип>)переменная
```

Пример:

```
int a      = 14;          // a равно 14
float c    = (float)a;    // c равно 14.0
```

Функции printf, scanf – синтаксис

`int printf(<format>, <аргумент 1>, <аргумент 2>, … );`

Функция printf преобразует, форматирует и выводит свои аргументы в стандартный поток вывода согласно строке формата – `format`.

Строка формата содержит два типа объектов: обычные символы, которые копируются в поток вывода, и спецификации формата, которые позволяют преобразовывать и выводить каждый из аргументов функции. Строка формата имеет вид

`%<флаги(необ.)><ширина(необ.)><.точность(необ.)><размер(необ.)>тип`

Флаги

Знак	Название знака	Значение	В отсутствие этого знака	Примечание
-	дефис	выводимое значение выравнивается по левому краю в пределах минимальной ширины поля	по правому	
+	плюс	всегда указывать знак (плюс или минус) для выводимого десятичного числового значения	только для отрицательных чисел	
	пробел	помещать перед результатом пробел, если первый символ значения не знак	Вывод может начинаться с цифры.	Символ **+** имеет больший приоритет, чем пробел. Используется только для десятичных числовых значений.
#	октоторп	«альтернативная форма» вывода значения		см. ниже
0	ноль	дополнять поле до ширины, указанной в поле *ширина* управляющей последовательности, символом 0	дополнять пробелами	Используется для типов d, i, o, u, x, X, a, A, e, E, f, F, g, G. Для типов d, i, o, u, x, X, если *точность* указана, этот флаг игнорируется. Для остальных типов поведение не определено.

Ширина

`Ширина` (десятичное число или символ звёздочка) указывает минимальную ширину поля (включая знак для чисел). Если представление величины больше, чем ширина поля, то запись выходит за пределы поля (например, %2i для величины 100 даст значение поля в три символа), если представление величины менее указанного числа, то оно будет дополнено (по умолчанию) пробелами слева, поведение может меняться предшествующими флагами. Если в качестве ширины указана звёздочка, ширина поля указывается в списке аргументов перед значением для вывода (например, `printf( "%0*i", 8, 4 );` выведет текст `00000004`).

Спецификатор точности

- указывает на минимальное количество символов, которое должно появиться при обработке типов d, i, o, u, x, X;
- указывает на минимальное количество символов, которое должно появиться после десятичной запятой (точки) при обработке типов a, A, e, E, f, F;
- максимальное количество значащих символов для типов g и G;
- максимальное число символов, которые будут выведены для типа s;

Точность задаётся в виде точки с последующим десятичным числом или звёздочкой (*), если число или звёздочка отсутствует (присутствует только точка), то предполагается, что число равно нулю. Точка для указания точности используется даже в том случае, если при выводе чисел с плавающей запятой выводится запятая.

Если после точки указан символ «звёздочка», то при обработке строки форматирования значение для поля читается из списка аргументов. (При этом, если символ звёздочка и в поле ширины и в поле точности, сначала указывается ширина, потом точность и лишь потом значение для вывода).

Пример

```
printf( "%0*.*f", 8, 4, 2.5 ); // выведет текст 00000002.5000
```

Спецификатор размера

Поле `размер` позволяет указать размер данных, переданных функции. Необходимость в этом поле объясняется особенностями передачи произвольного количества параметров в функцию в языке Си: функция не может «самостоятельно» определить тип и размер переданных данных, так что информация о типе параметров и точном их размере должна передаваться явно.

- h – если число следует вывести как короткое (short)
- l – если число следует вывести как длинное (long)

Тип

Символ	**Тип аргумента и способ вывода**
d, i	int; десятичное число
o	int; восьмеричное число
x, X	int; шестнадцатеричное число (abcdef при %x, ABCDEF при %X)
u	int; десятичное целое без знака
c	int; отдельный символ
s	char *; символы из строки, пока не 0, или в количестве, заданном параметром точности
f	double; десятичный вид
e, E	double; экспоненциальный вид
g, G	double; эквивалентно %e,%E если показатель точности меньше -4, или больше, или равен заданной точности. Иначе эквивалентно %f
p	void *; указатель (представление зависит от системы и языка)
%	Преобразование не выполняется, выводится %

Для ввода данных с преобразованием их в заданный формат используется функция scanf.

```
int scanf(<format>, <аргумент 1>, <аргумент 2>, … );
```

Функция `scanf` считывает данные с клавиатуры под управлением строки формата – `format` и присваивает преобразованные значения аргументам, **каждый из которых должен быть указателем**.

Пример

```
int g;
char f[20];
scanf("%d %s", &g, f);
```

Безусловные циклы. Инварианты циклов

Переменная, не меняющая в цикле – инвариантна. Если в цикле условие инвариантно, значит цикл безусловный.

Пример

```
while (1){ //Безусловный цикл
…
}
```

Проверка границ массивов.

В языке С не предусмотрена проверка за выход массива. Выход за границы массива (попытка чтения/записи несуществующего элемента) может привести к непредсказуемым результатам и поведению программы. Это является одной из самых распространенных ошибок.

Копирование массивов

`void * memcpy ( void * destination, const void * source, size_t num )` – копирует кусок памяти длиной *num*, начиная с указателя *source* в *destination.* Возвращает *destination*.

Операция копирования (присвоения) для структур

В языке С копирование структур выполняется копированием участка данных, описывающих одну структуру в адрес, описывающий другую структуру. А в языке С++ каждый член структуры копируется отдельно.

Рекурсия

Рекурсия — вызов функции из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия), например, функция B вызывает функцию A, а функция A — функцию B. Количество вложенных вызовов функции называется *глубиной рекурсии*.

```
int factorial(int f){
      if (f==1)
            return 1;
      return f * factorial(f-1);
}
```

Шаги, выполняемые при вызове `factorial(3)`

1. Вызов `factorial(3)`
2. `f != 1`, следовательно записываем текущий `f=3`, умножаем его на результат `factorial(2)`
3. `f != 1`, следовательно записываем текущий `f=2`, умножаем его на результат `factorial(1)`
4. `f == 1`, возвращаем 1
5. завершаем шаг 3, зная, что результатом `factorial(1)` является 1
6. завершаем шаг 2, зная, что результатом `factorial(2)` является 2
7. Возвращаем 6

Все шаги можно описать одним предложением: результатом вызова функции факториал будет 1, если параметром функции является 1, иначе, результатом является перемножение текущего параметра на результат вызова этой же функции для параметра, уменьшенного на 1.