Краткие ответы на экзаменационные вопросы по курсу “Системы программирования”

1. **Этапы жизненного цикла программного продукта.**
   - Анализ: определение, формализация, документирование требований к будущему программному продукту
   - Проектирование: формирование архитектуры системного программного продукта (декомпозиция: алгоритмы + модули, объекты + интерфейс)
   - Реализация: кодирование (непосредственное написание кода по описанной в спецификации функциональности и интерфейсам) + тестирование (запуск программы на некоторых входных данных с целью обнаружить дефект – несоответствие ожидаемого результата и выданного программой) + отладка (обнаружение причины дефекта и её исправление)
   - Внедрение (перенос продукта на целевую систему с инструментальной)
   - Сопровождение: эволюция программного продукта при его эксплуатации

2. **Состав и схема функционирования классической системы программирования.**
   - Редакторы текстов
   - Компиляторы, макрогенераторы, ассемблеры
   - Библиотеки
   - Редакторы связей
   - Загрузчики
   - Средства конфигурации
   - Отладчики
   - Средства тестирования
   - Профилировщики
   - Справочные системы
   - Системы планирования и координации работы

   Транслятор – программа, которая позволяет писать программу на языке высокого уровня и обеспечивает ее выполнение (позволяет писать не в машинных кодах)
   Компилятор - программа, которая осуществляет перевод исходной программы на ЯП в объектную программу (одна из разновидностей транслятора)
   Интерпретатор – так же 1 из разновидностей трансляторов, выход – результат обработки исходных данных
   Редактор связей - программа, получающая на вход отдельные файлы с неразрешёнными внешними ссылками и увязывающая отдельные связи
   Загрузчик - программа, обеспечивающая настройку модуля на конкретные адреса оперативной памяти, функции загрузчика в последнее время выполняет не система программирования, а операционная система
   Схема:
   Редактор текстов -> (исходная программа на некотором языке программирования) -> компилятор (+макрогенератор) -> (объектная программа – машинно-ориентированное представление программы, где есть информация в машинном коде + инф о внешних связях + таблицы констант…) -> редактор связей –> (исполняемый модуль – программа в машинных кодах как единое целое, относительно некоторого условного начального адреса) –> загрузчик (+отладчик)

3. **Типы трансляторов, особенности интерпретаторов и компиляторов.**
   - Компиляторы: выдают результат в виде исполняемого файла (в данном случае считаем, что компоновка входит в компиляцию). Этот файл транслируется один

раз (может быть запущен самостоятельно) и не требует для работы наличия на машине создавшего его транслятора

- Интерпретаторы: исполняют программу после разбора (в этом случае в роли объектного кода выступает внутреннее представление программы интерпретатором). Исполняется она построчно. В данном случае программа транслируется(интерпретируется) при каждом запуске (если объектный код кэшируется, возможны варианты) и требует для исполнения наличия на машине интерпретатора и исходного кода

Схема работы «чистого» компилятора: (исходная программа) -> компилятор -> (объектный модуль) -> (+ входные данные) -> (результаты работы)
Схема работы чистого интерпретатора: (исходная программа + входные данные) -> интерпретатор -> (результаты работы)
Смешанная стратегия: (исходная программа) -> компилятор промежуточного представления -> (промежуточное представление программы в байт-коде или машинном коде + вх. данные) -> интерпретатор промежуточного представления -> (результаты работы)

4. Общая схема работы компилятора.

Фаза анализа: (исходная программа на ЯП) -> лексический анализатор -> (последовательность лексем) -> синтаксический анализатор -> (промежуточное представление) -> семантический анализатор
Фаза синтеза: подготовка к генерации -> генерация кода -> (объектный модуль)

5. Основные понятия теории формальных грамматик и языков.

Алфавит - это конечное множество символов.
Цепочкой символов в алфавите V называется любая конечная последовательность символов этого алфавита.
Цепочка, которая не содержит ни одного символа, называется пустой цепочкой (обозначается ε)
Если α и β - цепочки, то цепочка $\alpha\beta$ называется конкатенацией (или сцеплением) цепочек α и β
Обращением (или реверсом) цепочки α называется цепочка, символы которой записаны в обратном порядке (обозначается α^R).
n-ой степенью цепочки α (будем обозначать α^n) называется конкатенация n цепочек α
Длина цепочки α - это число составляющих ее символов (обозначается $|\alpha|$)
Язык в алфавите V - это подмножество цепочек конечной длины в этом алфавите.
V* - множество, содержащее все цепочки конечной длины в алфавите V, включая пустую цепочку ε.
V^+ - множество, содержащее все цепочки конечной длины в алфавите V, исключая пустую цепочку ε.
Порождающая грамматика – это четверка (VT, VN, P, S), где

VT – алфавит терминальных символов (терминалов)
VN – алфавит нетерминальных символов (нетерминалов), не пересекающийся с VT
P – конечное подмножество множества $(VN \cup VN)^+ \times (VT \cup VN)^*$; элемент (α, β) называется правилом вывода и записывается в виде α -> β; в α есть хотя бы один нетерминал
S – начальный символ (цель) грамматики, S из VN

Цепочка $\beta \in (VT \cup VN)^*$ выводима из цепочки $\beta \in (VT \cup VN)^+$ в грамматике G = (VT, VN, P, S) (обозначим $\alpha \Rightarrow \beta$), если существуют цепочки $\gamma_0, \gamma_1, \dots, \gamma_n$, $(n \geq 0)$, такие, что $\alpha = \gamma_0$ -> γ_1 -> … -> $\gamma_n = \beta$
Последовательность $\gamma_0, \gamma_1, \dots, \gamma_n$ называется выводом длины n.

Языком, порождаемым грамматикой G = (VT, VN, P, S), называется множество $L(G) = \{\alpha \in VT^* \mid S \Rightarrow \alpha\}$.

Цепочка $\alpha \in (VT \cup VN)^*$, для которой $S \Rightarrow \alpha$, называется сентенциальной формой в грамматике G = (VT, VN, P, S).

6. **Эквивалентные грамматики.**

Грамматики G1 и G2 называются эквивалентными, если L(G1) = L(G2).

Грамматики G1 и G2 почти эквивалентны, если языки, ими порождаемые, отличаются более, чем на ε.

7. **Классификация формальных грамматик и языков по Хомскому.**

Тип 0: грамматики типа 0: на правила вывода не накладывается никаких ограничений, кроме ограничений по определению (в α есть хотя бы 1 нетерминал)

Тип 1: a) неукорачивающие грамматики: $\alpha \to \beta$, $|\alpha| \le |\beta|$, $\alpha \in (VN \cup VN)^+$, $\beta \in (VN \cup VN)^+$

b) контекстно-зависимые (КЗ): $\alpha \to \beta$, $\alpha = \xi^1 A \xi^2$; $\beta = \xi^1 \gamma \xi^2$; $A \in VN$, $\gamma \in (VN \cup VN)^+$; $\xi^1, \xi^2 \in (VN \cup VN)^*$

Тип 2: a) контекстно-свободные (КС): $A \to \beta$, $A \in VN$, $\beta \in (VN \cup VN)^+$

b) укорачивающие контекстно-свободные (УКС): $A \to \beta$, $A \in VN$, $\beta \in (VN \cup VN)^*$

Тип 3: a) праволинейные: A -> tB || A -> t; A, B∈VN, t∈VT

b) леволинейные: A -> Bt|| A -> t; A, B∈VN, t∈VT

8. **Соотношения между типами грамматик.**
   - Любая регулярная грамматика – КС и УКС
   - Любая КС – УКС, КЗ, неукорачивающая
   - Любая КЗ – типа 0
   - Любая неукорачивающая – типа 0
   - Любая УКС – типа 0
   - УКС, содержащая правила вида A->ε, не является КЗ и не является укорачивающей.

9. **Соотношения между типами языков.**

Язык L(G) является языком типа К, если его можно описать грамматикой типа К и нельзя грамматикой типа К + 1

- Любой регулярный язык – КС-язык, но не все КС-языки являются регулярными ($L=\{a^n b^n | n>0\}$)
- Любой КС-язык – КЗ-язык, но не все КЗ-языки являются КС-языками ($L=\{a^n b^n c^n | n>0\}$)
- Любой КЗ-язык - язык типа 0
- УКС-язык, содержащий ε, не является КЗ-языком

10. **Задача разбора. Дерево вывода.**

Вывод цепочки $\beta \in VT^*$ из $S \in VN$ в КС-грамматике G = (VT, VN, P, S), называется левым (левосторонним), если в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого левого нетерминала

Вывод цепочки $\beta \in VT^*$ из $S \in VN$ в КС-грамматике G = (VT, VN, P, S), называется правым (правосторонним), если в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого правого нетерминала

Дерево является деревом вывода(деревом разбора) в грамматике G = (VT, VN, P, S), если:

- каждая вершина дерева помечена символом из множества ($VN \cup VT \cup \varepsilon$), при этом корень дерева помечен символом S; листья - символами из ($VT \cup \varepsilon$)
- если вершина дерева помечена символом $A \in VN$, а ее непосредственные потомки - символами $a_1, a_2, \ldots, a_n$, где каждое $a_i \in (VT \cup VN)$, то $A \to a_1 a_2 \ldots a_n$ - правило вывода в этой грамматике;
- если вершина дерева помечена символом $A \in VN$, а ее единственный непосредственный потомок помечен символом ε, то $A \to \varepsilon$ - правило вывода в этой грамматике.

11. Неоднозначность грамматик и языков.

Грамматика G является неоднозначной, если существует $\alpha \in L(G)$, для которой существует более одного дерева вывода. В противном случае грамматика называется однозначной.

Проблема определения однозначности грамматики алгоритмически неразрешима.

Язык, порождаемый грамматикой, называется неоднозначным, если он не может быть порождён никакой однозначной грамматикой.

12. Недостижимые и бесполезные (бесплодные) символы грамматики. Алгоритмы удаления недостижимых и бесполезных (бесплодных) символов. Приведенная грамматика.

Символ $x \in (VT \cup VN)$ называется недостижимым в грамматике G = (VT, VN, P, S), если он не появляется ни в одной сентенциальной форме этой грамматики.

Алгоритм удаления недостижимых символов:

1. $V_0 = \{S\}$; i = 1
2. $V_i = \{x \mid x \in (VT \cup VN)$, в P есть $A \to \alpha x \beta$ и $A \in V_{i-1}$, a, $b \in (VT \cup VN)^*\} \cup V_{i-1}$
3. Если $V_i \neq V_{i-1}$, то i=i+1 и переходим к шагу 2, иначе $VN' = V_i \cap VN$; $VT' = V_i \cap VT$; P' состоит из правил множества P, содержащих только символы V_i; G' = (VT', VN', P', S).

Символ A из VN называется бесплодным в грамматике G = (VT, VN, P, S), если множество $\{a \in VT^* \mid A \to a\}$ пусто

Алгоритм удаления бесплодных символов:

Рекурсивно строим множества $N_0, N_1, \ldots$

1. $N_0 = 0$, i = 1
2. $N_i = \{A \mid (A \to \alpha) \in P$ и $\alpha \in (N_{i-1} \cup VT)^*\} \cup N_{i-1}$
3. Если $N_i \neq N_{i-1}$, то i=i+1 и переходим к шагу 2, иначе $VN' = N_i$, P' состоит из правил множества P, содержащих только символы из $VN' \cup VT$; G' = (VT, VN', P', S)

Грамматика называется приведенной, если в ней нет недостижимых и бесплодных символов.

Алгоритм приведения грамматики:

1. Обнаруживаются и удаляются все бесплодные символы
2. Обнаруживаются и удаляются все недостижимые символы

13. Определение недетерминированного конечного автомата (НКА).

Недетерминированный конечный автомат (НКА) - это пятерка (K, VT, F, H, S), где:

K – конечное множество состояний

VT – конечное множество допустимых входных символов

F – функция переходов: отображение множества $K \times VT \to K$

$H \subset K$ – конечное множество начальных состояний

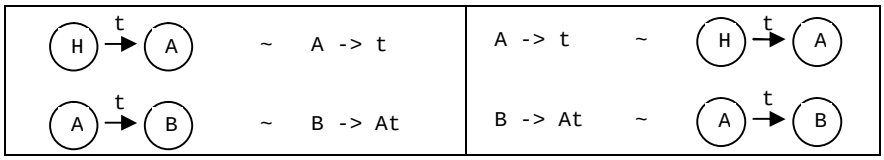
$S \subset K$ – конечное множество заключительных состояний

14. Диаграмма состояний (ДС) конечного автомата .

Диаграмма состояний (ДС) НКА – это ориентированный помеченный граф такой, что:

1. Его вершины помечены символами состояний из K
2. Вершины A и B соединяются дугой от A к B, если $\exists$ a$\in$VT: F(A,a)=B, при этом дуга помечется всеми такими a.

15. Леволинейные регулярные грамматики и конечные автоматы.



16. Определение детерминированного конечного автомата (ДКА).

Конечный автомат (НКА) - это пятерка (K, VT, F, H, S), где:

K – конечное множество состояний

VT – конечное множество допустимых входных символов

F – функция переходов: отображение множества K×VT -> K

H$\in$K – начальное состояние

S$\subset$K – конечное множество заключительных состояний

Конечный автомат называют детерминированным конечным автоматом (ДКА), если в каждом из его состояний для любого входного символа функция перехода содержит не более одного состояния: для любого a из VT и A из K: либо F(A, a) = {R}, R$\in$Q, либо P(a, q) = $\varnothing$

17. Алгоритм построения детерминированного конечного автомата по НКА.

M = (K, VT, F, H, S) – НКА; M' = (K', VT, F', H', S') – ДКА, допускающий тот же язык, что и M.

1. Множество состояний K' состоит из всех подмножеств множества K. Каждое состояние из K' будем обозначать $[A_1A_2\ldots A_n]$, где $A_i\in K$
2. Отображение F' определим как $F'([A_1A_2\ldots A_n], t) = [B_1B_2\ldots B_m]$, где для каждого $1 \le j \le m$ $F(A_i, t) = B_j$ для каких-либо $1 \le i \le n$
3. Пусть $H = \{H_1, H_2, \ldots H_k\}$, тогда $H' = [H_1, H_2, \ldots, H_k]$
4. Пусть $S = \{S_1, S_2, \ldots, S_p\}$, тогда S' – все состояния из K', имеющие вид $[\ldots S_i \ldots]$, $S_i\in S$ для какого-либо $1 \le i \le p$.

18. Задачи лексического анализа.

- Выделить в исходном тексте цепочку символов, представляющую лексему
- Удалить пробельные символы и комментарии
- Зафиксировать в специальных таблицах для хранения разных типов лексем факт появления соответствующих лексем в анализируемом тексте
- Преобразовать цепочку символов, представляющих лексему, в пару (тип лексемы, указатель на информацию о ней)

Лексический анализ важен для процесса компиляции по нескольким причинам:

- Замена в программе идентификаторов, констант, ограничителей и служебных слов лексемами делает представление программы более удобным для дальнейшей обработки
- Лексический анализ уменьшает длину программы, устраняя из ее исходного представления несущественные пробелы и комментарии
- Если будет изменена кодировка в исходном представлении программы, то это отразится только на лексическом анализаторе.

19. Лексический анализ на основе регулярных грамматик.

Лексемы можно описать с помощью регулярных грамматик.

Например, идентификатор (I): I -> a | b| ...| z | Ia | Ib |...| Iz | I0 | I1 |...| I9; целое без знака (N): N -> 0 | 1 |...| 9 | N0 | N1 |...| N9 и т.д.

Для грамматик этого класса, как мы уже видели, существует простой и эффективный алгоритм анализа того, принадлежит ли заданная цепочка языку, порождаемому этой грамматикой. Однако перед лексическим анализатором стоит более сложная задача: он должен сам выделить в исходном тексте цепочку символов, представляющую лексему, а также преобразовать ее в пару (тип_лексемы, указатель_на_информацию_о_ней). Для того, чтобы решить эту задачу, опираясь на способ анализа с помощью диаграммы состояний, введем на дугах дополнительный вид пометок - пометки-действия D_i. Смысл прежний - если в состоянии A очередной анализируемый символ совпадает с t_i для какого-либо i = 1, 2 ,... n, то осуществляется переход в состояние B; при этом необходимо выполнить действия D_1, D_2, ... ,D_m.

20. Объектная модель лексического анализатора. Схема его работы.

*****

21. Задачи синтаксического анализа.

- Проверка правильности синтаксиса
- Фиксация распознанной синтаксической структуры программы.

Для описания языка программирования достаточно грамматики типа 2.

Существуют алгоритмы зависимости сложности вычислений cn^3, cn^2.

Универсального алгоритма сложности cn нет. Есть только для формальной грамматики – метод рекурсивного спуска. Он лежит в основе многих методов, применяется к узкому подклассу КС грамматик.

22. Метод рекурсивного спуска (МРС): назначение, семантика процедур рекурсивного спуска.

Один из алгоритмов анализа входной цепочки, расходующий линейное время. Последовательность разбора эквивалентна построению дерева разбора методом «сверху вниз». Для каждого нетерминала грамматики создается своя процедура, носящая его имя, ее задача – начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала. Если такую подцепочку считать не удается, то процедура завершает свою работу вызовом процедуры обработки ошибки, которая выдает сообщение о том, что цепочка не принадлежит языку, и останавливает разбор. Если цепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляет возврат в точку вызова. Тело каждой такой процедуры пишется непосредственно по правилам вывода соответствующего нетерминала: для правой части каждого правила осуществляется поиск подцепочки, выводимой из этой правой части. При этом терминалы распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена.

23. Достаточные условия применимости метода рекурсивного спуска.

- Либо A->α, где а $\alpha \in (VN \cup VN)^*$ и это единственное правило вывода для этого нетерминала
- Либо A-> $a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n$, где $a_i \in VT$ для всех i = 1, 2, …, n; $a_i \neq a_j$ при $i \neq j$; $\alpha_i \in (VN \cup VN)^*$, т. е. если для нетерминала A правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть различимыми.

24. Исследование применимости МРС в случае наличия ε-альтернативы и итерационных правил.

1. МРС заведомо не применяется, если в грамматике есть правила, заведомо не рекурсивные, т.к. постоянно обращаются к A( ). Получается зацикливание. Например,
   A -> Aα | β
   Преобразования:
   A -> βB
   B -> αB | ε
   При этом B должен быть новым терминальным символом в грамматике; ε обязательно, иначе преобразование не равносильно.

2. Две альтернативы начинаются с одинакового терминального символа:
   A -> $a\alpha_1$ | $a\alpha_2$ | β
   Тогда этот символ выносится и вводится новый нетерминальный символ.
   Преобразования:
   A -> aB | β
   B -> α_1 | α_2

3. Есть альтернативы, начинающиеся с терминальных символов.
   A -> Bα | β
   Так как мы рассматриваем приведённые грамматики, то существует правило вида B -> γ_1 | γ_2 , которое раскрывает B.
   Преобразования:
   A -> $\gamma_1\alpha$ | $\gamma_2\alpha$ | β
   см. пункт 2, если γ_1 и γ_2 начинаются с одного символа.
   В результате избавились от нетерминальных символов в правой части.

4. A -> Aα | β | ε
   Это УКС грамматика. Если есть правило с ε, метод применим не всегда.
   Преобразования:
   S -> βAα
   A -> αA | ε

   ```
   void A( ) {
           if (c=='α') { gc( ); A( ); }
   }
   ```

   Например, для цепочки βαα метод неприменим: в правиле S -> βAα при выходе из A( ) мы должны считать α, но перед этим мы уже считали символ конца ввода, поэтому метод неприменим.

Пусть FIRST (A) – множество терминальных символов, с которых начинаются цепочки, выводимые из этого терминального символа; FOLLOW (A) – множество терминальных символов, с которых начинаются подцепочки, следующие за данным нетерминальным символом A.
Правило: Если $FIRST(A) \cap FOLLOW(A) \neq \varnothing$, то метод не применим, если $= \varnothing$, тогда данное правило не влияет на применимость метода (эти множества надо считать лишь для тех нетерминальных символов, из которых выводится ε).

Пример.
S -> fASd | ε
A -> Aa | Ab | dB | f

B -> bcB | ε

1. Всегда сначала надо избавиться от левой рекурсии (A -> Aa | Ab)

=> S -> fASd | ε
A -> dBA’ | fA’
A’ -> aA’ | bA’ | ε
B -> bcB | ε

2. После первого преобразования видно, что надо избавляться от ε-правил.
   а) FIRST (S) = {f}; FOLLOW (S) = {d}; пересечение - пустое множество => первое ε-правило не мешает
   б) FIRST (A’) = {a,b}; FOLLOW (A’) = {f,d}; пересечение пусто
   в) FIRST (B) = {b}; FOLLOW (B) = {a,b,f,d}; в пересечении получается {b} => МРС не применим => надо преобразовывать

3.

=> S -> fASd | ε
A -> dB’ | fA’
B’ -> bcB’ | A’
A’ -> aA’ | bA’ | ε

4. B’ -> bcB’ | A’ - не подходит к МРС, т.к. альтернатива начинается нетерминальным символом.

=> S -> fASd | ε
A -> dB’ | fA’
B’ -> bcB’ | aA’ | bA’ | ε
A’ -> aA’ | bA’ | ε

=> S -> fASd | ε
A -> dB’ | fA’
B’ -> bC | aA’ | ε
C -> cB’ | A’ - необходимо переписать
A’ -> aA’ | bA’ | ε

=> S -> fASd | ε
A -> dB’ | fA’
B’ -> bC | aA’ | ε
C -> cB’ | aA’ | bA’ | ε
A’ -> aA’ | bA’ | ε

S ~ S

После всех преобразований надо опять проверить.
FIRST (S) = {f}; FOLLOW (S) = {d}; Λ = 0
FIRST (B’) = {a,b}; FOLLOW (B’) = {f,d} ; Λ = 0
FIRST (A’) = {a,b}; FOLLOW (A’) = {f,d}; Λ = 0
FIRST (C) = {a,b,c}; FOLLOW (C) = {f,d}; Λ = 0
- грамматика преобразована к виду, к которому применим метод рекурсивного спуска.

25. Задачи семантического анализа. Грамматики с действиями.

Проверка контекстных условий – семантический анализ:

- Каждый используемый а программе идентификатор должен быть описан, но не более одного раза в одной зоне описания

- При вызове функций число фактических параметров и их типы должны соответствовать числу и типам формальных параметров
- Обычно в языке накладываются ограничения на типы операндов любой операции, определенной в этом языке, на типы левой и правой части в присваивании, на тип параметра цикла, на тип условия в операторе цикла и условном операторе, и т п

Необходимо расширить грамматику и вставить необходимые действия.
Семантический анализ реализуется МРС, и им же вводятся дополнительные действия.

26. Объектная модель синтаксического анализатора.
*****

27. Использование исключений C++ при обработке синтаксических ошибок и нарушении контекстных условий.
Для обработки синтаксических ошибок и нарушений контекстных условий очень удобно пользоваться механизмом исключений C++. Для этого необходимо:

i. Вместо вызовов функции ERROR() во время рекурсивного спуска писать throw с параметром, идентифицирующим тип ошибки (например, номер ошибки при условии существования пронумерованного списка ошибок),
ii. Запуск РС-метода поместить в try-блок,
iii. В соответствующий ему catch-блок поместить обработку этих ошибок.

Выгода использования исключений заключается в простоте реализации, т.е. в том, что нет необходимости реализовывать выход из рекурсии при возникновении ошибки.

28. Свойства языка внутреннего представления программы, примеры таких языков.

a. Он позволяет фиксировать синтаксическую структуру исходной программы
b. Текст на нем можно автоматически генерировать во время синтаксического анализа
c. Его конструкции должны просто транслироваться в объектный код, либо достаточно эффективно интерпретироваться

Некоторые общепринятые способы внутреннего представления программ:

- Постфиксная запись
- Префиксная запись
- Многоадресный код с явно именуемыми результатами
- Многоадресный код с неявно именуемыми результатами
- Связные списочные структуры, представляющие синтаксическое дерево

29. Синтаксически управляемый перевод: идея, принципы организации, примеры.
В основе – грамматика с действиями. Параллельно с анализом исходной цепочки лексем выполняются действия по генерации внутреннего представления программы. Для этого грамматика дополняется соответствующими процедурами генерации.

E -> T { + T }	E -> T { + T <putchar('+')> }
T -> F { * F }	T -> F { * F >putchar ('*')> }
F -> a | b | (E)	F -> a <putchar('a')>|b<putchar('b')| (E)

30. ПОЛИЗ выражений.
ПОЛИЗ выражений задаётся следующими правилами:

- Если E является простым (единственным) операндом, то его ПОЛИЗ - это и есть этот операнд E,
- ПОЛИЗом выражения E1 * E2, где * - любая бинарная операция, а E1 и E2 - её операнды, является запись E1' E2' *, где E1' и E2' - ПОЛИЗ запись выражений E1 и E2 соответственно,

- ПОЛИЗом выражения * E, где * - любая унарная операция, а E - её операнд, является запись E' *, где E' - ПОЛИЗ запись выражения E,
- ПОЛИЗом выражения (E) является ПОЛИЗ выражения E

31. ПОЛИЗ операторов языков программирования.

Для определения ПОЛИЗа операторов ЯП, необходимо ввести дополнительные обозначения. Пусть:

- $\underline{I}$ – означает, что операндом является адрес переменной I, а не её значение,
- Для реализации ПОЛИЗа уловных операторов введём операции условного и безусловного перехода:
  - Пусть ПОЛИЗ оператор, помеченный меткой L находится на позиции p (будем считать, что все элементы ПОЛИЗ-записи пронумерованы). Тогда оператор безусловного перехода **goto** L в ПОЛИЗ будет записываться так: p! – где ! – это оператор ПОЛИЗ
  - Для реализации условного перехода введём переход по лжи: **if(not** B**) goto** L. Также пусть оператор, помеченный L, стоит на позиции p. Тогда запись этого условного перехода будет выглядеть так: B' p !F, где !F – оператор ПОЛИЗ, а B' – это ПОЛИЗ-запись выражения B
- Операторы ввода/вывода в ПОЛИЗ обозначаются одноместными операциями. Пусть R – обозначение операции ввода, а W – обозначение операции вывода.

Используя введённые операции ПОЛИЗ, приведём некоторые примеры ПОЛИЗа операторов M-языка программирования:

- Оператор присваивания
  I := E **ó** $\underline{I}$, E, :=
- Условный оператор
  if E then S1 else S2 **ó** P(E), L1, !F, P(S1), L2, !, [L1] P(S2) [L2]
- Оператор цикла while-do
  while E do S **ó** [L2] P(E), L1, !F, P(S), L2, ! [L1]
- Оператор цикла do-while
  do S while E **ó** [L2] P(S), P(E), L1, !F, L2, ! [L1]
- Оператор цикла for
  for(A; B; C) do S **ó** P(A), [L3] P(B), L1, !F, L2, !, [L4] P(C), L3, !, [L2] P(S), L4, ! [L1]
- Оператор ввода read(I) **ó** $\underline{I}$ R
- Оператор вывода write(E) **ó** P(E) W

32. Генерация ПОЛИЗа выражений и операторов.

Каждый элемент в ПОЛИЗе – это лексема, то есть пара вида (*номер_класса, номер_в_классе*). Для этого расширим набор лексем:

- Будем считать, что операции !, !F, R, W относятся к ограничителям языка, наряду с остальными операциями
- Для описания ссылок на элемент ПОЛИЗа введём тип 0, то есть элемент вида (0, p) обозначает ссылку на элемент под номером p
- Для операндов-адресов введём тип 5. То есть операнды-значения (идентификаторов) имеют тип 4, а их адреса – тип 5.

Генерация ПОЛИЗа происходит во время синтаксического анализа параллельно с контролем контекстных условий, поэтому для генерации можно использовать информацию, «собранную» синтаксическим и семантическим анализаторами. Например, при генерации ПОЛИЗа выражений можно воспользоваться стеком для проверки типов выражений или добавить операции генерации ПОЛИЗа в функцию для этой проверки.

33. Интерпретация ПОЛИЗа.

ПОЛИЗ просматривается слева направо, если встречаем операнд, то записываем его в стек, если встретили знак операции, то извлекаем из стека нужное количество операндов и выполняем операцию, результат (если он есть) записываем в стек.

34. Использование исключений C++ при обработке ошибок периода выполнения.
******

35. Основные стратегии распределения памяти.

Явное выделение блоков фиксированного размера, которые связываются в список, над которым достаточно легко выполнять операции выделения и освобождения. Плюсы – если программа полностью использует блок памяти, то никаких дополнительных расходов не нужно. С каждым блоком связан указатель на его начало, и надо хранить только информацию о том, занят блок или нет. Свободные блоки можно «подшивать» между собой, используя часть блока для хранения указателей, с помощью которых осуществляется объединение свободной памяти в список. Когда очередной блок свободной памяти выделяется программе, он удаляется из списка, а начальный указатель принимает значение указателя на следующий свободный блок памяти. Когда какой-то блок памяти освобождается, он добавляется в список свободной памяти.

Достоинства: простота.

Недостатки: какая-то часть памяти используется неэффективно в случае, если по размеру требуется меньше, чем размер блока.

Явное выделение блоков переменного размера. Один из методов выделения блоков переменного размера – метод первого подходящего. При выделении блока размера s находится первый блок размера f >= s. Затем этот блок разбивается на два – с размерами s и f-s. Но мы можем и не найти такой фрагмент. Тогда необходимо приостанавливать выполнение программы и искать все использованные фрагменты и перемещать их на дно кучи

Достоинства: эффективное использование памяти, место не тратится попусту.

Недостатки: дольше работает программа.

Неявное выделение/освобождение памяти. Неявно выделяемые блоки памяти также могут быть фиксированного или переменного размера. При неявном динамическом выделении и освобождении блоков памяти, выделяемые блоки обычно имеют следующую структуру: - размер блока (для блоков переменного размера)

- счетчик ссылок, пометка (обычно есть либо одно, либо другое)
- указатели на блоки
- то, что досталось пользователю, заказавшему этот блок

Счетчик ссылок подсчитывает количество указателей в программе, которые ссылаются на этот блок, если счетчик равен 0, то блок не используется и его можно освободить. Существует проблема циклических ссылок, когда счетчик всегда >= 0.

Пометка фиксирует задействован ли блок или нет, то есть имеется ли у программы хотя бы 1 указатель, ссылающийся на этот блок. В некоторый момент начинает работать сборщик мусора. Он помечает все блоки как недостижимые, а потом начинает анализ текущих указателей программы. Блоки, на которые ничего не указывает, считаются свободными и их можно перегруппировать.

36. Принципы реализации виртуальных функций.
*****

37. Машинно-независимая оптимизация и машинно-зависимая оптимизация. Примеры оптимизирующих преобразований.

- Машинно-независимые преобразования:

1. Удаление недостижимого кода
   if (1) S1; else s2 => s1;
2. Оптимизация линейных участков программы:
   a) Удаление бесполезных присваиваний
      a=b*c; d=b+c; a=d*c; => d=b+c; a=d*c;
   b) Исключение избыточных вычислений:
      d=d+b*c; a=d+b*c; c=d+b*c; => t=b*c; d=d+t; a=d+c; c=a;
   c) Свёртка объектного кода. Производится во время компиляции только для тех операций, для которых операнды уже известны.
      i=2+1; j=6*i+i; => i=3; j=21;
   d) Перестановка операций:
      a=2*b*3*c; => a=(2*3)*(b*c);
      a=(b+c)+(d+c); => a=(b+(c+(d+c)));
   e) Арифметические преобразования:
      a=b*c+b*d; => a=b*(c+d);
      a*1 => a; a*0 => 0; a+0 => a;
   f) Оптимизация вычисления логических выражений:
      a || b || c || d => a, если a=true;
      Но: a || f(b) || g(c) - сохраняется, т.к функции могут иметь побочные эффекты.
1. Оптимизация передачи параметров в процедуры || функции.
   Обычно параметры передаются через стек, и на эту процедуру может тратиться очень много времени.
   a) Передача параметров через регистры. Но, помещая переменную в регистр, мы не можем использовать её адрес.
      В C++ есть специальный унификатор register, который ставится для разрешения помещения параметра в регистр.
   b) Подстановка кода функции (вместо вызова функции в объектный код – т.к. на вызов функции тратится время).
   Компиляторы могут это делать не только с макросами, но и с обычными фукциями, но только с разрешения пользователя.
4. Оптимизация циклов.
   a) Вынесение инвариантных вычислений из циклов
      for (i=1; i<=10; i++) a[i]=b*c*a[i]; => d=b*c; for (i=1; i<=10; i++) a[i]=d*a[i];
   b) Замен операций с индуктивными переменными (перменными, образующими арифметическую прогрессию).
      - for (i=1; i<=N; i++) a[i]=i*10; =>
        => t=10; i=1; while (i<=N) {a[i]=t; t=t+10; i++;}
      - S=10; for (i=1; i<=N; i++) {r=r+f(S); S=S+10; } =>
        => S=10; m=N*10; while (S<=m) {r=r+f(S); S=S+10; }
   c) Слияние и развёртывание циклов.
      Слияние:
      for (i=1; i<=N; i++) for (j=1; j<=M; j++) a[i][j]=0; =>
      => K=m*N; //(остаётся 1 цикл)
            for (i=1; i<=k; i++) a[i]=0;
      Развёртывание:
      for (i=1; i<=3; i++) a[i]=i; => a[1]=1; a[2]=2; a[3]=3;

- Машинно-зависимые преобразования:

1. Распределение регистров процессора.
2. Оптимизация кода для процессора, допускающая распараллеливание вычислений. (В программе надо выделить куски кода, эти куски вычисляются независимо друг от друга => их можно вычислять параллельно по разным процессам.)
   a+b+c+d+e+f
   1 поток. => ((((a+b)+c)+d)+e)+f (без распараллеливания)
   2 потока. => ((a+b)+c)+((d+e)+f)
   3 потока. => (a+b)+(c+d)+(e+f)

38. Интегрированная среда разработки (ИСР).

ИСР объединила в себе возможности текстовых редакторов исх. текстов программ и командный язык компиляции. Пользователь не должен выполнять всю последовательность действий от порождения исходного кода программы до его выполнения, от него также не требуется описывать makefile. Достаточно только удобной интерфейсной форме указать состав исходных модулей и библиотек. Ключи, необходимые компилятору и др. техническим средствам, также задаются в виде интерфейсных форм настройки.

Содержит в себе:

- Репозиторий – организованное хранилище информации, появляющейся в течение всего “жизненного цикла” создания программного продукта.
- Специальные автоматические средства разработки образов. Например, языки 4 поколения (4GL), оперирующие образами. Такие средства позволяют осуществлять разделение обработки (создания) программы между несколькими разработчиками.
- Редакторы текстов.
- Средства документирования.
- Средства тестирования и отладки.
- Средства управления.
- Средства реинжениринга (т.е. восстановления структуры программы по коду).

Примеры ИСР – TurboPascal, Delphi, Visual Studio, K-Develop.

Ключевые особенности:

- Интегрированность среды
- Библиотека компонент
- Визуальная технология разработки
- Технология two-ways-tool
- Поддержка работы с базами данных
- «горячие клавиши»
- X-курсор
- Останов с редактированием, пошаговое выполнение, подсветка выполняемой строки

39. Основные функции редактора текста в рамках ИСР. Примеры его интегрированности с другими компонентами ИСР.

1. Подготовка текста программы.
2. Многооконный интерфейс с поддержкой “буксировки” текста мышкой (функция drag & drop – перенос фрагмента мышкой).
3. Интеграция с компилятором.
   a) Визуализация текста в выделении лексем.

b) Дополнение кода (интерактивная подсказка).
   Например,
   a**.** …… (A – класс) - после выполнения такой команды получим список, что входит в "a".
   f (…… - дополнение кода или интерактивная подсказка.
c) Шаблоны кода – часто используемые фрагменты программы.
d) Всплывающие подсказки.
e) Выделение места, в котором при компиляции обнаружена ошибка.

4. Интеграция с отладчиком.
   a) Отображение контрольных точек останова при отладке.
   b) Отображение текущего значения объекта при наведении курсора на идентификатор.

40. Отладчики, их возможности. Примеры интегрированности отладчика с другими компонентами ИСР.

- Пошаговое выполнение программы (шаг = строка, с трассировкой внутри вызываемой функции или без нее)
- Выполнение программы до строки, в которой в редакторе стоит курсор
- Выделение выполняемой строки в данный момент
- Приостановка выполнения программы
- Можно запросить значение переменной
- Можно заказать вычисление некоторого выражения
- Можно изменить значение переменной и продолжить выполнение программы
- Расставить/снять точки останова, которые визуализируются в текстовом редакторе
- Вся информация должна выдаваться в терминах исходной программы

41. Редактор внешних связей, его назначение и принципы работы. Загрузчик.

- Он должен разрешить межмодульные связи (для объектных файлов, порождаемых компилятором при раздельной трансляции модулей, составляющих программу)
- Должен связать объектные файлы, порожденные компилятором, и библиотечные файлы, входящие в состав системы программирования (для статически связываемых библиотек)

Загрузчик обеспечивает подготовку готовой программы к выполнению, обрабатывают ресурсы, полученные с выхода компиляторов. Модуль, выполняющий преобразование относительных адресов в абсолютные непосредственно в момент запуска программы на выполнение.

42. Библиотеки. Основные типы библиотек.

a. Библиотеки функций - определяют возможности СП в целом, чем больше функций, тем лучше. Подразделяются на 2 класса:
   - библиотеки для языков программирования
   - библиотеки для решения задач какой-то проблемной области

Библиотеки функций представляют собой библиотеки откомпилированных объектных модулей.

b. Библиотеки классов – важная часть СП, базируются на объектно-ориентированных языках программирования. Основной недостаток – все классы должны быть написаны на том же языке, что и программа.
   - конкретные классы
   - абстрактные классы
   - шаблоны классов

Существует так называемая проблема “жирного интерфейса” – возникает желание включать в библиотеки больше функций, но, с другой стороны, нельзя допускать и перегрузки.
Интерфейсными называются функции, входящие в public-часть класса.
Библиотеки классов компилируются вместе с программой.

c. Библиотеки компонент – готовые откомпилированные программные модули.
В настоящее время используются следующие технологии:
- CORBA – исполняемые программные компоненты из сети. Существует её реализация для большинства систем. Технология не зависит от используемого языка.
- COM – исполняемые программные компоненты, размещённые локально (на компьютере пользователя). Модифицированные версии COM – DCOM, ActivX.
- Java Beans – исполняемые программные компоненты на языке Java.

Нельзя путать библиотеки с пакетами прикладных программ!
Пакеты прикладных программ (ППП) – специальным образом организованные программные комплексы, используемые для определённой области деятельности.
Программу, написанную в виде ППП, нельзя включить в свою программу, а программу из библиотеки – можно.

Для подключения статических библиотек включаются файлы, на уровне редактора связей подключаются конкретные тела.

Динамически подключаемые библиотеки подключаются не при компиляции, а в процессе выполнения. Редактор связей формирует некоторую точку вызова подключаемой библиотеки. Существует некоторая группа команд, вызывающая функции данной библиотеки. Преимущества динамически подключаемых библиотек:

- не требуется включать код часто используемых функций
- несколько программ могут использовать код одной библиотеки
- нет необходимости перекомпилировать свои программы при изменении текста программы в библиотеке.

В виде динамических библиотек оформлены системные функции, например, API.

43. Критерии проектирования стандартных библиотек.

1. Общезначимость содержимого
2. Эффективность
3. Безопасность – не должны допускать провоцирование ошибок, а, наоборот, предотвращать их
4. Завершённость
5. Сочетаемость с базовыми типами данных
6. Возможность служить фундаментом для других библиотек

44. Стандартная библиотека C++.

Обеспечивает:

1. поддержку свойств языка
   - управление памятью – new/delete.
   - предоставление информации о типах во время выполнения программы.
   - поддержка обратных исключений – те библиотечные средства, которые используются при запуске программы.
2. предоставление информации о зависящих от реализации аспектах языка.
3. предоставление общеупотребительных функций.

4. предоставление некоторых нетривиальных и машинно-зависимых средств (например, ввод/вывод, сортировка при работе со списком)

Имена локализованы в пространстве имён std – т.е. можно использовать эти имена, и это не будет приводить к конфликту.
При записи <cstdio> буква “c” обозначает, что файл подключается именно из стандартной библиотеки C.

45. Стандартная библиотека шаблонов STL: контейнеры, итераторы, алгоритмы, аллокаторы.

Контейнеры.

Контейнер – шаблонный класс для хранения объектов какого-либо одного и того же типа. Контейнеры бывают различных типов. Например, в классе vector определяется динамический массив, queue – очередь, list – линейный список. Помимо таких базовых контейнеров, в библиотеке стандартных шаблонов определены и ассоциативные контейнеры, которые позволяют получать хранящиеся в них значения. Например, в классе map определён ассоциативный список, доступ к элементам которого осуществляется с помощью уникальных ключей. Т.е. в таком списке элемент – это пара «значение-ключ».

Контейнеры STL:

Vector <T> – динамический массив.
List <T> – линейный список.
Stack <T> – стек.
Queue <T> – очередь.
Deque <T> – двусторонняя очередь.
Priority_queue <T> – очередь с приоритетом.
Set <T> – множество с уникальными элементами.
Bitset <N> – множество битов.
Multiset <T> – множество не обязательно с уникальными элементами.
Map <key, val> – ассоциативный список, в котором хранятся пары ключ/значение, с каждым ключом связано одно значение.
Multimap <key, val> – ассоциативный список, в котором хранятся пары ключ/значение, с каждым ключом связано не обязательно одно значение.

Основные типы:

Следующие типы определены в public-части для каждого контейнера:
value_type
allocator_type - распределитель памяти
size_type
iterator, const_iterator
reverse_iterator, const_reverse_iterator
pointer, const_pointer - указатель на элементы
reference, const_reference - ссылка на элементы

Распределитель памяти.

У каждого контейнера есть свой распределитель памяти allocator, который управляет процессом выделения памяти для объектов контейнера. По умолчанию распределитель памяти – это объект класса allocator. Также можно написать свой распределитель памяти и использовать его в программах, если STL не используем.

Рассматриваем только интерфейс функций:

```
template <class T> class allocator {
……
public:
typedef  T*pointer;
typedef  T&reference;
……
allocator( ) throw( );
//- конструктор по умолчанию
……
pointer allocate (size_type n);
}
```

По умолчанию берётся стандартный распределитель памяти, он берёт память из области динамической памяти.

Алгоритмы.

<algorithm>

В STL имеется 60 алгоритмов. Алгоритмы реализуют некоторые распространённые операции с контейнерами, которые не включены в контейнер. Т.е алгоритм – шаблонная функция, поэтому их можно использовать с контейнерами любых типов.

Алгоритмы подразделяются на 3 группы:

- немодифицирующие алгоритмы – действия сводятся к просмотру контейнера или выделению из него какой-либо функции:
  find ( ); - поиск элемента, выдаёт место, на котором находится элемент.
  count ( ); - подсчёт
  for_each( );
- модифицирующие:
  transform ( );
  reverse ( );
  copy ( ); - копируется содержимое одного контейнера в другой
- сортировки:
  sort ( ); - простая сортировка
  stable_sort( ); - сортировка, гарантирующая сохранение относительно порядка элементов
  merge( ); - слияние двух списков.

Итераторы.

Итераторы – это что-то вроде указателей на элементы контейнера.

Типы итераторов перечислены в контейнере. В public-части контейнера есть итераторные функции с одинаковыми именами.

Объекты класса итератора играют роль указателей для контейнера. Итераторы используются для доступа к содержимому контейнера примерно так же, как указатели для доступа к элементам массива.

Итераторный класс определяется внутри контейнера.

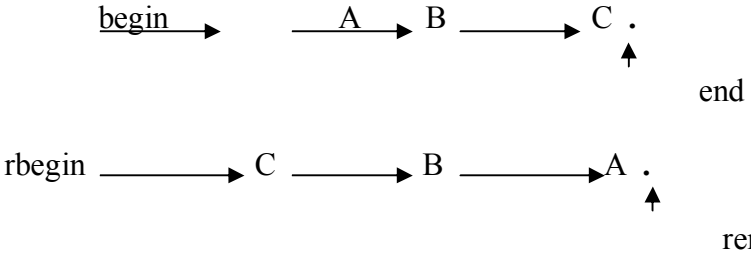
Следует обратить внимание, что при использовании нужно писать оператор расширения видимости, чтобы знать, к какому итератору относится написанное:

```
list <int> :: iterator
```

Для итераторов нет константы NULL, обозначающей пустой указатель.

reverse_iterator - обратный итератор, выдаёт последовательность в обратном порядке.

Различие между этими итераторами:



Типы итераторов:

Существуют следующие типы операторов, которые позволяют соответствующие функции:

1. Итераторы вывода.
   *p_ , p++
2. Итераторы ввода.
   *=p, ->, ++, ==, !=
3. Однонаправленные.
   *p=, =*p, ->, ++, ==, !=
   - положить в контейнер
   - взять из контейнера
4. Двунаправленные.
   *p=, =*p, -> , ++, ==, !=, --
5. С произвольным доступом.
   *p=, =*p, -> , ++, ==, !=, --, [ ], +, -, +=, -=, <, >, <=, >=

С 1 до 5 увеличивается мощность итераторов.

46. Стандартная библиотека шаблонов STL: шаблонные классы vector и list.

Контейнер vector (схема):

```
template <class T, class A = allocator <T>> class vector {
                                - параметр распределения памяти, по умолчанию
берётся
                                  стандартный распределитель памяти
……..
public:
        //типы
        //итераторные функции
        //доступ к элементам
/*const*/reference operator [ ] (size_type n);        //доступ без контроля выхода за
диапазон вектора
/*const*/reference at (size_type n);          //с контролем
/*const*/reference front ( );                 //указатель на первый элемент
контейнера
/*const*/reference back ( );                  // указатель на последний элемент
контейнера
explicit vector (const A& = A( ));
                        // - здесь опускается имя формального параметра, работает
конструктор по умолчанию
                // explicit vector - вектор нулевой длины (когда данные записываются
                в вектор, сначала
                // создаётся такой вектор)
explicit vector (site_type n; const T& value = T( ); const A& = A( ));
```

…… // здесь определён конструктор копирования, оператор присваивания и др.

```
// функции – члены класса, частые в использовании
iterator erase (iterator i);
iterator insert (iterator i, const T& value = T( ));
                    // - вставка перед этим элементом, возвращает итератор вставленного элемента
void push_back ( );
void pop_back ( );     // - удаляем последнюю запись вставленного элемента
sizetype size ( )  const;
bool empty ( )  const;
void clear ( );          // - очищаем вектор
…….
}
```

Контейнер list (схема):

```
template <class T, class A = allocator <T>> class list {
public:
        //типы
        //итераторные функции
/*const*/reference front ( );                  //указатель на первый элемент контейнера
/*const*/reference back ( );                   // указатель на последний элемент контейнера
explicit list (const A& = A( ));
explicit list (site_type n; const T& value = T( ); const A& = A( ));
        //explicit – для безопасности, не позволяет делать присваивание векторов.
……

iterator erase (iterator i);
iterator insert (iterator i, const T& value = T( ));
                    // - вставка перед этим элементом, возвращает итератор вставленного
                        элемента
void push_back ( );
void pop_back ( );     // - удаляем последнюю запись вставленного элемента
sizetype size ( )  const;
bool empty ( )  const;
void clear ( );          // - очищаем вектор
…….
}
```