1. Понятие программной инженерии, ее цели и задачи. Стандарты программной инженерии.

Программная инженерия является отраслью информатики (computer science) и изучает вопросы построения компьютерных программ, обобщает опыт программирования в виде комплекса общих знаний и правил регламентации инженерной деятельности разработчиков программного обеспечения

2. Основные этапы разработки программ, их назначение и характеристики.

Этапы	
1. Постановка задачи	8. Выполнение
2. Выбор метода решения	9. Тестирование
3. Разработка алгоритма	10. Отладка
4. Написание программы на языке программирования	11. Документирование
5. Ввод программы в компьютер	12. Эксплуатация
6. Трансляция	13. Сопровождение
7. Компоновка	14. Снятие с эксплуатации

Постановка задачи:

Цель: определение функциональных возможностей программы, подготовка технического задания и внешней спецификации (достаточно полная и точная формулировка решаемой задачи)

Написание программы на языке программирования

Язык программирования — формальная знаковая система, предназначенная для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и семантических правил, задающих внешний вид программы и действия, которые выполнит исполнитель (компьютер) под ее управлением

- Программа – логически упорядоченная последовательность команд, необходимых для решения определенной задачи
- Текст программы – полное законченное и детальное описание алгоритма на языке программирования

Программа – алгоритм, записанный на языке программирования

Трансляция — это преобразование программы с одного языка программирования в семантически эквивалентный текст на другом языке

Компоновка - это процесс сборки программы из объектных модулей, в котором производится их объединение в исполняемую программу и связывание вызовов внешних функций и их внутреннего представления (кодов), расположенных в различных объектных модулях. При этом могут объединяться один или несколько объектных модулей программы и объектные модули, взятые из библиотечных файлов и содержащие стандартные функции и другие инструкции

Выполнение программы

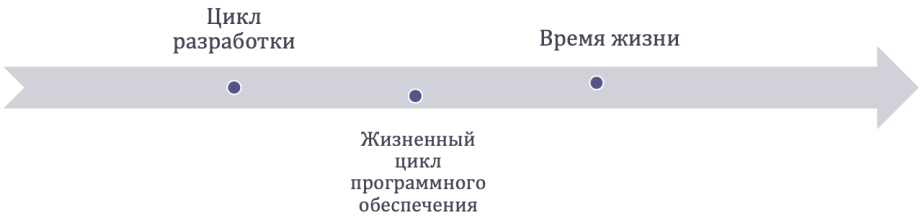
- *Исполняемый файл* — это файл, который может быть обработан или выполнен компьютером без предварительной трансляции
- *Формат исполняемого файла* — это соглашение о размещении в нём машинных команд и вспомогательной информации

Тестирование – это процесс исполнения программы с целью обнаружения ошибок

Отладка — этап разработки компьютерной программы, в ходе выполнения которого обнаруживают, локализуют и устраняют ошибки

3. Порядок прохождения задач через ЭВМ. Назначение и результаты каждого этапа.

4. Жизненный цикл программного обеспечения. Основные понятия, участвующие в определении жизненного цикла. Структура жизненного цикла ПО согласно стандарта ISO/IEC 12207: 1995.



Жизненный цикл программного обеспечения – это период времени, который начинается с момента принятия решения о необходимости создания программного обеспечения и заканчивается в момент его полного изъятия из эксплуатации

- **Артефакты** — создаваемые человеком информационные сущности – документы, в достаточно общем смысле участвующие в качестве входных данных и получающиеся в качестве результатов различных деятельностей.
- **Роль -** абстрактная группа заинтересованных лиц, участвующих в деятельности по созданию и эксплуатации системы, решающих одни и те же задачи или имеющих одни и те же интересы по отношению к ней
- **Программный продукт** – набор компьютерных программ, процедур и, возможно связанных с ними документации и данных
- **Процесс** – совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные

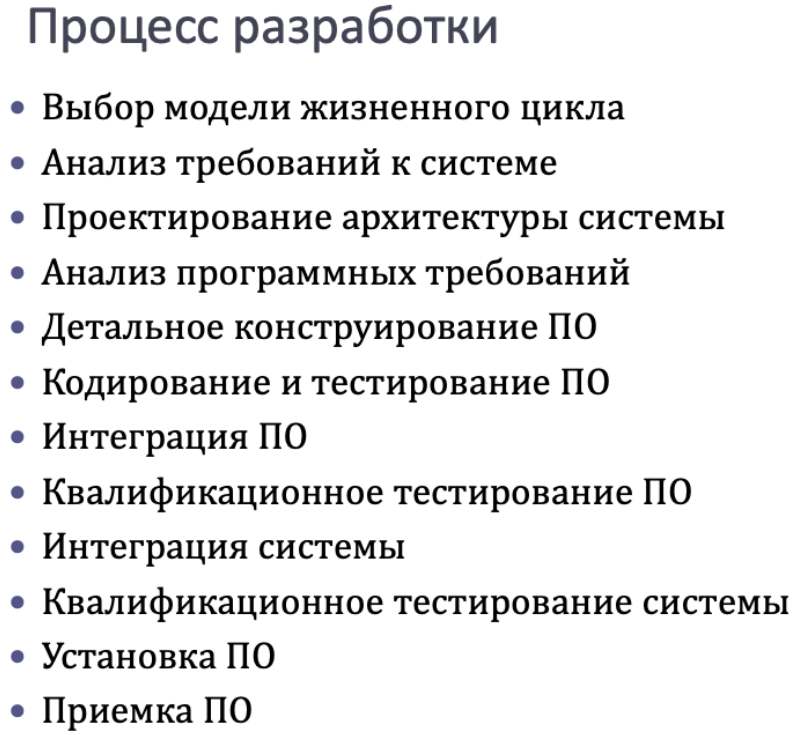
5. Основные процессы жизненного цикла программного обеспечения, их роль в создании ПО и краткое описание. Процесс разработки ПО в соответствии со стандартом ISO/IEC 12207: 1995.

Основные процессы цикла:

- Приобретение – определят действия заказчика, приобретающего ПО. Процесс получения программного продукта или программной услуги. Заказчик: осознает свои потребности в программной системе; подготавливает заявочные предложения, содержащие требования к системе
- Постановка – определяет действия организации-поставщика по отношению к заявочным предложениям заказчика. Отвечает за выполнение процессов разработки, эксплуатации и / или сопровождения. Включает: рассмотрение заявочных предложений; подготовка договора с заказчиком; планирование

выполнения работ; разработку организационной структуры проекта, технических требований к среде разработки и ресурсам, мероприятий по управлению проектом

- Разработка – определяет действия разработчика в процессе создания ПО и его компонентов в соответствии с заданными требованиями. Включает: оформление проектной и эксплуатационной документации; подготовку материалов, необходимых для проверки работоспособности программного продукта; разработку материалов для обучения / подготовки персонала
- Эксплуатация – установление эксплуатационных стандартов и проведение эксплуатационного тестирования
- Сопровождение – внесение изменений в ПО для исправления ошибок, повышения производительности, адаптации к изменчивым условиям работы



6. Вспомогательные и организационные процессы жизненного цикла программного обеспечения. Их назначение и связь с основными процессами.

Организационные процессы: (для создания и реализации основной структуры, охватывающей взаимосвязанные процессы жизненного цикла и соответствующий персонал, а также для постоянного совершенствования данной структуры и процессов)

- Управление проектом
- Создание инфраструктуры
- Оценка и улучшение жизненного цикла
- Обучение

Вспомогательные процессы: (является целенаправленной составной частью другого процесса, обеспечивающей успешную реализацию и качество выполнения программного проекта)

- Документирование
- Управление конфигурацией
- Обеспечение качества
- Верификация
- Аттестация
- Совместная оценка
- Аудит
- Разрешение проблем

Связь ->

7. Качество программного обеспечения и пути его достижения. Виды качества. Категории метрик, используемые при измерении качества программного обеспечения. Методы контроля качества ПО.

категорийные, или описательные (номинальные) метрики используются для оценки функциональных возможностей программных средств;

количественные метрики применимы для измерения надежности и эффективности сложных комплексов программ;

качественные метрики в наибольшей степени соответствуют практичности, сопровождаемости и мобильности программных средств

- Методы и техники, связанные с выяснением свойств ПО во время его работы. Это прежде всего все виды *тестирования*, а также измерение количественных показателей качества, которые можно определить по результатам работы ПО – эффективности по времени и другим ресурсам, надежности, доступности и пр.
- Методы и техники определения показателей качества на основе симуляции работы ПО с помощью моделей разного рода. К этому виду относятся *проверка на моделях* (model checking), а также *прототипирование* (*макетирование*), используемое для оценки качества принимаемых решений
- Методы и техники, нацеленные на выявление нарушений формализованных правил построения исходного кода ПО, проектных моделей и документации. К методам такого рода относится инспектирование кода, заключающееся в целенаправленном поиске определенных дефектов и нарушений требований в коде на основе шаблонов, автоматизированные методы поиска ошибок в коде, не основанные на его выполнении, методы проверки документации на согласованность и соответствие стандартам
- Методы и техники обычного или формализованного анализа проектной документации и исходного кода для выявления их свойств. К этим методам относятся многочисленные методы анализа архитектуры ПО, методы формального доказательства свойств ПО и формального анализа эффективности применяемых алгоритмов

8. Характеристики, используемые для оценки качества программного обеспечения (в соответствии со стандартом ISO/IEC 9126).

Функциональные возможности (Functionality)	Способность программного обеспечения реализовать установленные или предполагаемые потребности пользователей
Надежность (Reliability)	Способность программного обеспечения сохранять свой уровень качества функционирования при установленных условиях за установленный период времени
Практичность (Usability)	Характеризуется объемом работ, требуемых для использования программного обеспечения определенным или предполагаемым кругом пользователей
Эффективность (Efficiences)	Определяется соотношением между уровнем качества функционирования программного обеспечения и объемом используемых ресурсов при установленных условиях
Сопровождаемость (Maintainability)	Характеризует объем работ, требуемых для проведения конкретных изменений (модификаций)
Мобильность (Portability)	Способность программного обеспечения быть перенесенным из одного окружения в другое

9. Современные модели оценки качества программного обеспечения. Теоретические предпосылки построения моделей. Достоинства и недостатки каждой модели.

?

10. Системы управления версиями. Возможности, предоставляемые системой управлений версиями. Типичный цикл работы с проектом при использовании систем управления версиями.

---

11. Разработка алгоритмов. Свойства алгоритмов. Процесс алгоритмизации. Способы описания алгоритмов. Примеры.

Свойства алгоритмов

- Дискретность – возможность разбиения на шаги
- Понятность – ориентация на конкретного исполнителя
- Определенность – однозначность толкования инструкций
- Конечность – возможность получения результата за конечное число шагов
- Массовость – применимость к некоторому классу объектов
- Эффективность – оптимальность времени и ресурсов, необходимых для реализации алгоритма

Процесс алгоритмизации

- разложение всего вычислительного процесса на отдельные шаги – возможные составные части алгоритма, что определяется внутренней логикой самого процесса и системой команд исполнителя
- установление взаимосвязей между отдельными шагами алгоритма и порядка их следования, приводящего от известных исходных данных к искомому результату
- полное и точное описание содержания каждого шага алгоритма на языке выбранной алгоритмической системы
- проверка составленного алгоритма на предмет, действительно ли он реализует выбранный метод и приводит к искомому результату

Способы описания алгоритмов

- словесно-формульный (на естественном языке, вербальный)
- структурный или блок-схемный (графический)
- с использованием специальных алгоритмических языков (нотаций)
- с помощью сетей Петри
- программный

Пример

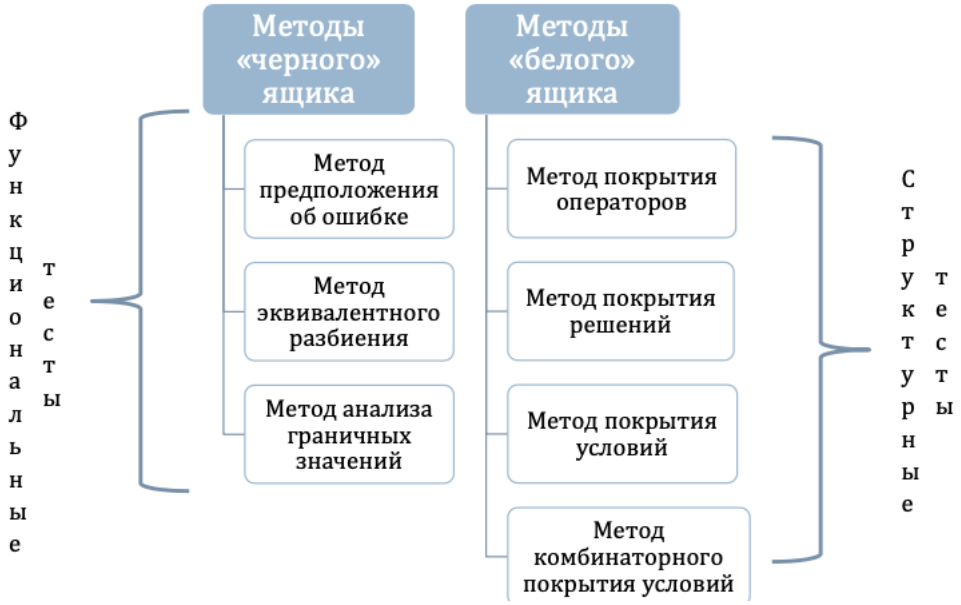
y=2a-(x+6)

Словесно-формульным способом алгоритм решения этой задачи может быть записан в следующем виде:

1. Ввести значения *a* и *x*
2. Сложить *x* и *6*
3. Умножить *a* на *2*
4. Вычесть из *2a* сумму (*x+6*)
5. Вывести y как результат вычисления выражения

12. Тестирование программ. Цели и задачи тестирования. Методы тестирования программного обеспечения.

Тестирование – это процесс исполнения программы с целью обнаружения ошибок



13. Тестирование «черным ящиком». Виды тестов (примеры). Метод эквивалентного разбиения. Классы эквивалентности тестов и способы их выделения (примеры). Построение тестов.

- Тестирование программы ограничивается использованием небольшого подмножества всех возможных входных данных
- Правильно выбранный тест этого подмножества должен обладать двумя свойствами:
  - уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели тестирования
  - покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных

Класс эквивалентности – это класс, в рамках которого все тесты являются эквивалентными, т.е. приводящими к одному и тому же результату

Способы

- правильные классы эквивалентности, представляющие правильные входные данные программы
- неправильные классы эквивалентности, представляющие все другие возможные состояния условий (т.е. ошибочные входные значения)

Построение тестов

- Назначение каждому классу эквивалентности уникального номера
- Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых правильных классов эквивалентности, до тех пор пока все правильные классы эквивалентности не будут покрыты тестами
- Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы эквивалентности не будут покрыты тестами

14. Тестирование «черным ящиком». Виды тестов (примеры). Метод анализа граничных значений и способы его применения (примеры). Метод предположения об ошибке. Проектирование и исполнение теста.

- Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных значений осуществляется таким образом, чтобы проверить тестом каждую границу этого класса
- При разработке тестов рассматривают не только входные условия (пространство входов), но и *пространство результатов* (т.е. выходные классы эквивалентности)

1. Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений

Например, если рассматривается правильная область входных значений от –1.0 до +1.0, то нужно написать тесты для ситуаций –1.0, +1.0, –1.001 и +1.001

2. Построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих значений, если входное условие удовлетворяет дискретному ряду значений

Например, если входной файл может содержать от 1 до 255 записей, то получить тесты для 0, 1, 255 и 256 записей

3. Использовать первое правило для каждого выходного условия

Например, если программа вычисляет ежемесячный расход и если минимум расхода составляет 0.00 руб., а максимум – 10000 руб., то надо построить тесты, которые вызывают расход в 0.00 руб. и 10000 руб. Кроме того, следует построить, если это возможно, тесты, которые вызывают отрицательный расход и расход больше 10000 руб.

Метод предположения об ошибке

Опытный программист часто интуитивно предполагает вероятные типы ошибок и затем разрабатывает тесты для их обнаружения

Например, если тестируется программа сортировки списка, то следует рассмотреть следующие специфические ситуации:

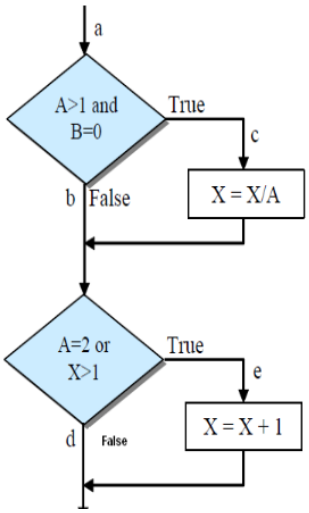
1. сортируемый список пуст
2. сортируемый список содержит только одно значение
3. все записи в сортируемом списке имеют одно и то же значение
4. список уже отсортирован

15. Стратегия разработки тестов. Тестирование методом «белого» ящика: покрытие решений и комбинаторное покрытие условий.

Стратегия разработки тестов

- Определить правильные и неправильные классы эквивалентности для входных и выходных данных и дополнить, если это необходимо, тесты, построенные на предыдущем шаге
- Провести анализ граничных значений
- Для получения дополнительных тестов рекомендуется использовать метод предположения об ошибке
- Проверить логику программы на полученном наборе тестов. Для этого нужно воспользоваться критерием покрытия решений, покрытия условий, либо комбинаторного покрытия условий, при необходимости подготовив недостающие тесты

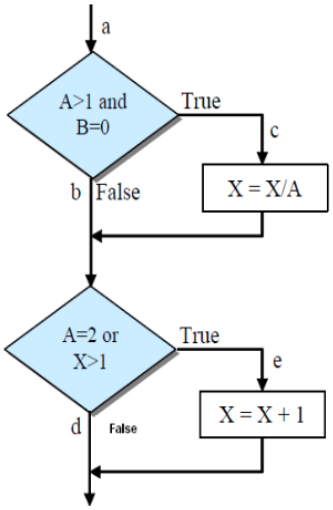
Стратегии «белого» ящика: покрытие решений



Покрытие решений требует, чтобы каждое решение имело результатом значения *истина и ложь* и при этом каждый оператор выполнялся бы, по крайней мере, один раз

Покрытие решений может быть выполнено двумя тестами, покрывающими либо пути *ace и abd* ***(A=2, B=0, X=3*** и ***A=3, B=1, X=2)***, либо пути *acd и abe* ***(A = 3, B = 0, X = 3*** и ***A = 2, B = 1, X = 1***)

Стратегии «белого» ящика: комбинаторное покрытие условий



Критерием полноты тестирования является создание такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении и все точки входа выполнялись, по крайней мере, один раз

1. A > 1, B = 0 — A = 2, B = 0, X = 4 условия 1 и 5 (путь ***ace***)
2. A > 1, B ≠ 0 — A = 2, B = 1, X = 1 условия 2 и 6 (путь ***abe***)
3. A ≤ 1, B = 0 — A = 1, B = 0, X = 2 условия 3 и 7 (путь ***abd***)
4. A ≤ 1, B ≠ 0 — A = 1, B = 1, X = 1 условия 4 и 8 (путь ***abd***)
5. A = 2, X > 1
6. A = 2, X ≤ 1
7. A ≠ 2, X > 1
8. A ≠ 2, X ≤ 1

Пропущен путь ***acd***

16. Отладка программ. Виды ошибок в программах и последовательность их обнаружения. Методы отладки.

Отладка — этап разработки компьютерной программы, в ходе выполнения которого обнаруживают, локализуют и устраняют ошибки

Ошибка – это расхождение между вычисленным, наблюдаемым и истинным, заданным или теоретически правильным значением

Классификация программных ошибок:

- синтаксические (нарушение грамматических правил языка программирования)
- семантические (нарушение порядка следования параметров функций, неправильное построение выражений)
- прагматические или логические (заключаются в неправильной логике алгоритма, нарушении смысла вычислений и т. п.)

- ошибки трансляции (компиляции):
  - ошибки соответствия синтаксису языка
  - ошибки компоновки (ошибки связи)
  - ошибки данных
- ошибки выполнения
- ошибки логики

- Отладка за столом:
  - просмотр
  - проверка
  - Прокрутка
- Программный способ отладки (отладочная печать):
  - эхо–печать входных данных
  - печать в ветвях программы
  - печать в узлах программы
- Аппаратный способ (встроенные интегрированные средства отладки):
  - выполнение по шагам
  - просмотр переменных в окне наблюдения
  - локализация места ошибки при выполнении программы до курсора

17. Структурное программирование. Два подхода к проектированию и разработке программ в рамках структурного программирования. Их достоинства и недостатки.

Восходящее тестирование

Достоинства	Недостатки
Восходящее тестирование	
Имеет преимущества, если ошибки сконцентрированы главным образом в модулях нижнего уровня	Необходимо разрабатывать автоматические модульные тесты
Легче создавать тестовые условия	Программа как единое целое не существует до тех пор, пока не добавлен последний модуль
Проще оценка результатов	

Нисходящее тестирование

Достоинства	Недостатки
Нисходящее тестирование	
Имеет преимущества, если ошибки сосредоточены главным образом в верхней части программы	Необходимо разрабатывать заглушки, которые часто оказываются сложнее, чем кажется вначале
Представление теста облегчается после подключения функции ввода-вывода	До применения функций ввода-вывода может быть сложно интегрировать тестовые данные в заглушки
Раннее формирование структуры программы позволяет провести ее демонстрацию пользователю и служит моральным стимулом для разработчиков	Может оказаться трудным или невозможным создать тестовые условия
	Сложнее производится оценка результатов тестирования
	Допускается возможность формирования представления о совмещении тестирования и проектирования
	Стимулируется незавершение тестирования некоторых модулей

18. Защитное программирование. Принципы защитного программирования. Рекомендации по реализации защитного программирования.

Защитное программирование предполагает такой стиль написания программ, при котором появляющиеся ошибки легко обнаруживаются и идентифицируются программистом

Принципы защитного программирования

- Общее недоверие к данным – входные данные каждого модуля должны тщательно анализироваться в предположении, что они ошибочны
- Немедленное обнаружение – каждая программная ошибка должна быть выявлена как можно раньше, что упрощает установление ее причины
- Изолирование ошибок – ошибки в одном модуле должны быть изолированы так, чтобы не допустить их влияния на другие модули

Рекомендации по реализации защитного программирования

- Проверяйте тип данных. Контролируйте буквенные поля (поля имен), чтобы убедиться, что они не содержат цифровых данных. Проверяйте цифровые поля на отсутствие в них буквенных данных
- Делайте проверку области значений переменных, чтобы удостовериться, например, что положительные числа всегда положительны
- Выполняйте контроль правдоподобности значений переменных, которые не должны превышать некоторых констант или значений других переменных. Например, начисляемые налоги и удержания не должны быть больше суммы, для которой они определяются
- Контролируйте итоги вычислений путем введения всюду, где это возможно, перекрестных итогов, контрольных сумм и счетчиков числа обрабатываемых элементов информации

19. Защитное программирование. Подходы к выбору метода обработки ошибки. Утверждения: примеры использования в программах. Условная компиляция, случаи применения, примеры.

Утверждение (assertion) — это код (метод или макрос), используемый во время разработки, с помощью которого программа проверяет правильность своего выполнения

void assert (int arg); (для использования необходимо подключить библиотеку assert.h)

Assert предназначен для документирования и проверки истинности допущений, сделанных при написании кода

```
#include <assert.h>
#include <stdio.h>

double Salory;        // Зарплата
double Tax;           // Налог
double Payment;    // Вознаграждение

   printf (“Введите зарплату:  “);
   scanf (“%lf”, &Salory);

   assert(Salory > 0) and (Salory < 100000);

   Tax=Salory*0.13;
   assert(Tax<Salory);
   printf (“Налог = %lf \n“ , Tax);
   Payment = Salory-Tax;
   assert(Payment <Salory);
   printf (“Сумма на руки = %lf \n”, Payment);
```

Условная компиляция

используется в том случае, когда удобно задать какую-то переменную и затем проверить, если эта переменная определена, то выполнить какую-то последовательность действий, а если нет, то не выполнять

Пример для С:

```
#define DEBUG
...
#if defined( DEBUG )
// отладочный код
...
#endif
```