

Основы программной инженерии (ПОИТ)  
Технологии разработки программного обеспечения (ИСиТ)

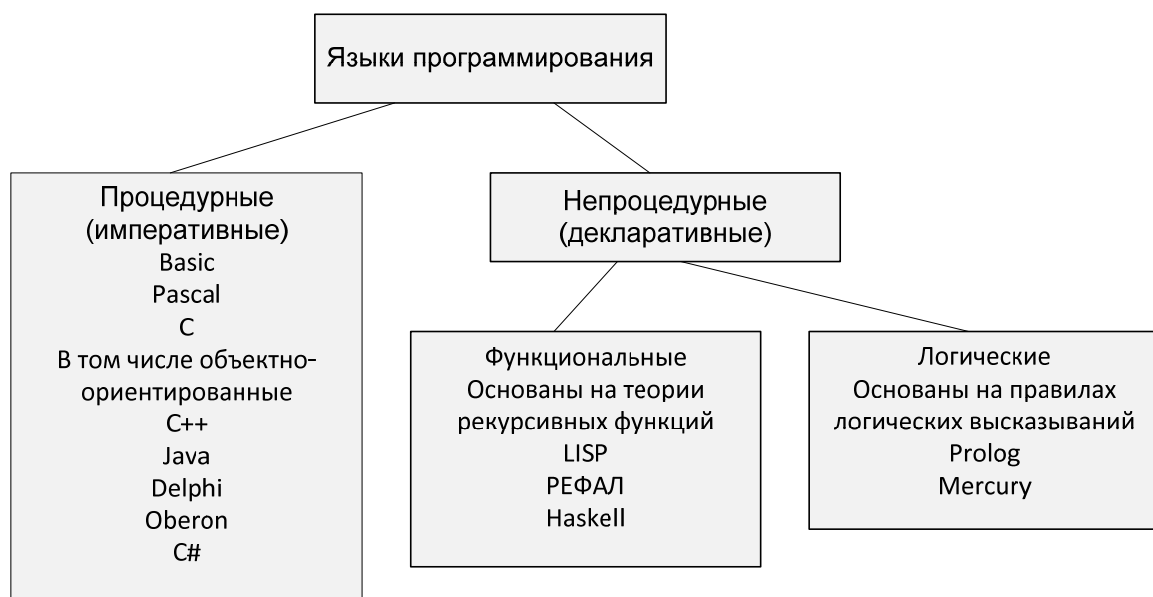
Стили программирования. Модульное программирование

План лекции:

- парадигмы программирования;
- императивное программирование;
- декларативное программирование;
- структурное программирование;
- модульное программирование
- стили программирования (оформление кода).

1. На прошлых лекциях:

Парадигмы (стили) программирования



Язык программирования строится в соответствии с базовой моделью вычислений и парадигмой программирования.

**Парадигма программирования** – это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию).

**Методология** включает в себя модель вычислений для данного стиля.

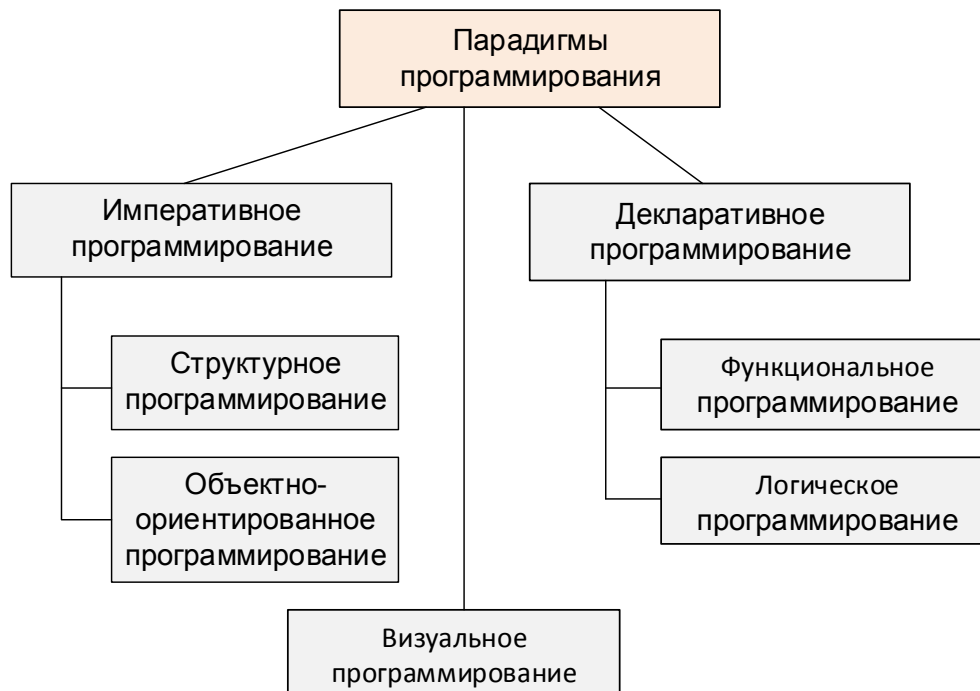
**Методология разработки** программного обеспечения – совокупность методов, применяемых на различных стадиях жизненного цикла программного обеспечения.

**Неструктурное программирование** характерно для наиболее ранних языков программирования. Сложилось в середине 40-х с появлением первых языков программирования.

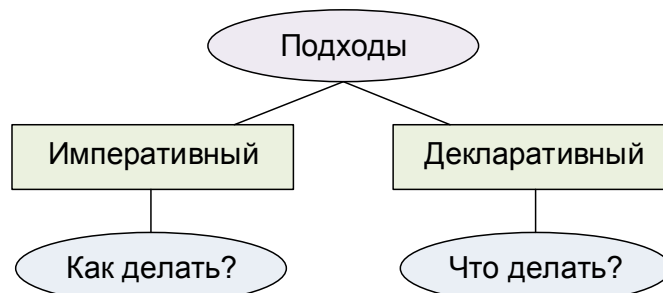
Основные признаки:

- строки как правило нумеруются;
- из любого места программы возможен переход к любой строке;

**Основные парадигмы программирования:**



**Основные подходы к программированию:**



**Императивное программирование** (от греч. imper — действие) предполагает, что программа явно описывает алгоритм решения конкретной задачи (действия исполнителя), т.е. описывает как решать поставленную задачу.

**Декларативное программирование** (лат. declaratio — объявление) — это предварительная реализация «**решателя**» для целого класса задач.

Тогда для решения *конкретной задачи* этого класса достаточно декларировать в терминах данного языка только её условие:

(исходные данные + необходимый вид результата)

«**Решатель**» сам выполняет процесс получения результата, реализуя известный ему алгоритм решения.

## Структурное программирование

**Структурное программирование** – методология и технология разработки программных средств, основанная на трёх базовых конструкциях:

- следование;
- ветвление;
- цикл.

### *Цели структурного программирования:*

- повысить надежность программ (улучшить структуру программы);
- создание понятной, читаемой программы, которая выполняется, тестируется, конфигурируется, сопровождается и модифицируется без участия автора (создание ПП).

### *Принципы разработки:*

- программирование «сверху-вниз» (нисходящее программирование);
- модульное программирование с иерархическим упорядочением связей между модулями/подпрограммами «От общего к частному»

### *Этапы проектирования:*

- формулировка целей (результатов) работы программы;
- представление процесса работы программы (модель);
- выделение из модели фрагментов: определение переменных и их назначения, стандартных программных контекстов.

**Технология структурного программирования** базируется на следующих методах:

- нисходящее проектирование (формализация алгоритма «сверху вниз»: движение от общего к частному);
- пошаговое проектирование (нисходящая пошаговая детализация программы);
- структурное проектирование (замена формулировки алгоритма на одну из синтаксических конструкций – *последовательность*, *условие* или *цикл*; программирование без goto);
- одновременное проектирование алгоритма и данных (процессы детализации алгоритма и введение данных, необходимых для работы, идут параллельно);
- *модульное проектирование*;
- *модульное, нисходящее, пошаговое тестирование*.

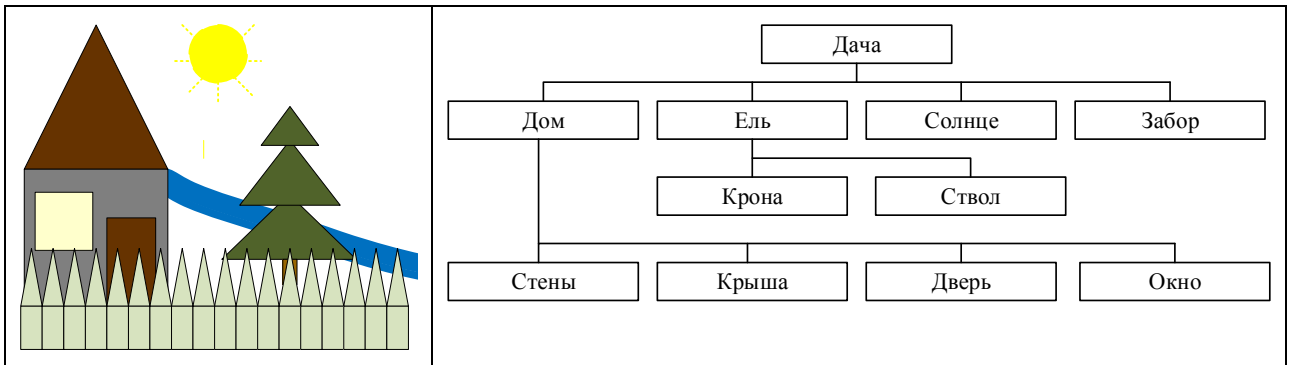
**<цель\_результата> = <действие> + <цель\_результата вложенной конструкции>**

Программирование **без goto**.

Использование операторов (вида **continue, break, return**) для более изменения структурированной логики выполнения программы.

## 2. Модульное программирование

«Разделяй и властвуй» - латинская формулировка принципа, лежащего в основе модульного проектирования.



**Модульное программирование** – это организация программы как совокупности небольших независимых блоков, называемых модулями.

**Модуль** – функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом.

**Функциональная декомпозиция задачи** – разбиение большой задачи на ряд более мелких, функционально самостоятельных подзадач – модулей.

Каждый модуль в функциональной декомпозиции представляет собой «черный ящик» с одним входом и одним выходом.

Модуль – это фрагмент описания процесса, оформленный как самостоятельный программный продукт, пригодный для многократного использования.



Профессор Никлаус Вирт

Никлаус Вирт — швейцарский учёный, специалист в области информатики, один из известнейших теоретиков в области разработки языков программирования, профессор компьютерных наук Швейцарской высшей технической школы Цюриха, лауреат премии Тьюринга

Создатель и ведущий проектировщик языков Pascal, Modula-2, Oberon, участвовал в разработке многих языков программирования.

Разработал язык Modula (1975 г.), в котором реализовал идеи модульного программирования с хорошо определенными межмодульными интерфейсами и параллельного программирования.

***Цели модульного программирования:*** уменьшить сложность программ; предотвратить дублирование кода, упростить тестирование программы и обнаружение ошибок.

В языках высокого уровня, как правило, реализация данного механизма выполняется путем разделения модулей на интерфейсную и реализующую части.

**Пример** проекта, состоящего из основного файла и одного модуля на C/C++:

Главный модуль (требуется подключить интерфейс модуля)

```
1  #include "hello.h"
2  int main()
3  {
4      hello();
5      return 0;
6  }
```

Заголовочный файл модуля hello  
(интерфейс - заголовочный файл  
hello.h)

```
1  #include <iostream>
2  void hello();
```

Файл реализации модуля hello  
(файл с реализацией hello.c)

```
1  #include "hello.h"
2  void hello()
3  {
4      printf("Hello!\n");
5  }
```

#### **Плюсы модульного программирования:**

- ускорение разработки (позволяет изменять реализацию функциональности модуля, не затрагивая при этом взаимодействующие с ним модули);
- повышение надежности (локализует влияние потенциальных ошибок рамками модуля);
- упрощение тестирования и отладки;
- взаимозаменяемость.

#### **Минусы модульного программирования:**

- модульность требует дополнительной работы программиста и определенных навыков проектирования программ.

### ***Модуль, основные характеристики:***

- один вход и один выход (на вход программный модуль получает набор исходных данных, выполняет их обработку и возвращает набор выходных данных);
- функциональная завершенность (модуль выполняет набор определенных операций для реализации каждой отдельной функции, достаточных для завершения начатой обработки данных);
- логическая независимость (результат работы данного фрагмента программы не зависит от работы других модулей);
- слабые информационные связи с другими программными модулями (обмен информацией между отдельными модулями должен быть минимален);
- размер и сложность программного элемента должна быть в разумных рамках.

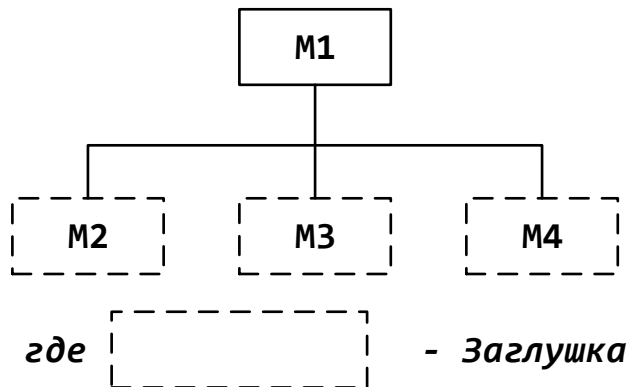
**Программный модуль** является самостоятельным программным продуктом. Это означает, что каждый программный модуль разрабатывается, компилируется и отлаживается отдельно от других модулей программы.

Роль модулей могут играть структуры данных, библиотеки функций, классы, сервисы и другие программные единицы, реализующие некоторую функциональность и предоставляющие интерфейс к ней.

### ***Технология модульного программирования*** базируется на следующих ***методах:***

- методы нисходящего проектирования (назначение – декомпозиция большой задачи на меньшие так, чтобы каждую подзадачу можно было рассматривать независимо.);
- методы восходящего проектирования.

### *Нисходящее проектирование программ и его стратегии.*



На первом этапе разработки кодируется, тестируется и отлаживается головной модуль, который отвечает за логику работы всего программного комплекса. Остальные модули заменяются заглушками, имитирующими работу этих модулей.

На последних этапах проектирования все заглушки постепенно заменяются рабочими модулями.

#### *Стратегии, на которой основана реализация:*

- пошаговое уточнение (данная стратегия разработана Е. Дейкстрой);
- анализ сообщений (данная стратегия базируется на работах группы авторов: Иодана, Константайна, Мейерса).

**Пример.** Стратегия, основанная на использовании псевдокода.

Способы реализации пошагового уточнения.

1. Кодирование программы с помощью псевдокода и управляющих конструкций структурного программирования;
2. Использование комментариев для описания обработки данных. Пошаговое уточнение требует, чтобы взаимное расположение строк программы обеспечивало читабельность всей программы.

#### *Преимущества* метода пошагового уточнения:

- основное внимание при его использовании обращается на проектирование корректной структуры программ, а не на ее детализацию;
- так как каждый последующий этап является уточнением предыдущего лишь с небольшими изменениями, то легко может быть выполнена проверка корректности процесса разработки на всех этапах.



**Недостаток** метода пошагового уточнения:

- на поздних этапах проектирования может возникнуть необходимость в структурных изменениях, которые повлекут за собой пересмотр более ранних решений.

### **Использование псевдокода.**

#### **Пример.**

Пусть программа обрабатывает файл с датами.

Необходимо отделить правильные даты от неправильных, отсортировать правильные даты, определить летние даты и вывести их в выходной файл.

**Первый этап** пошагового уточнения: *задается* заголовок программы, соответствующий ее основной функции.

**Program** Обработка\_дат

**Второй этап** пошагового уточнения: *определяются* основные действия.

**Program** Обработка\_дат;

Отделить\_правильные\_даты\_от\_неправильных {\*}

Обработать\_неправильные\_даты;

Сортировать\_правильные\_даты;

Выделить\_летние\_даты;

Вывести\_летние\_даты\_в\_файл;

**EndProgram.**

**Третий этап** пошагового уточнения: *детализация* фрагмента \*.

**Program** Обработка\_дат;

**While** не\_конец\_входного\_файла

**Do**

**Begin**

Прочитать\_дату;

Проанализировать\_правильные|неправильные\_даты;

**End**

Обработать\_неправильные\_даты;

Сортировать\_правильные\_даты;

Выделить\_летние\_даты;

Вывести\_летние\_даты\_в\_файл;

**EndProgram.**

и так далее.

### ***Метод восходящей разработки***

При восходящем проектировании разработка идет снизу-вверх.

На первом этапе разрабатываются модули самого низкого уровня.

На следующем этапе к ним подключаются модули более высокого уровня и проверяется их работоспособность.

На завершающем этапе проектирования разрабатывается головной модуль, отвечающий за логику работы всего программного комплекса.

Методы нисходящего и восходящего программирования имеют свои преимущества и недостатки.

#### ***Недостатки нисходящего проектирования:***

- необходимость заглушек;
- до самого последнего этапа проектирования неясен размер всего программного комплекса и его характеристики, которые определяются только после реализации модулей самого низкого уровня.

***Преимущество нисходящего проектирования*** – на самом начальном этапе проектирования отлаживается головной модуль (логика программы).

***Недостаток восходящего программирования*** – головной модуль разрабатывается на завершающем этапе проектирования, что порой приводит к необходимости дорабатывать модули более низких уровней.

***Преимущество восходящего программирования*** – не нужно писать заглушки.

На практике применяются оба метода. Метод нисходящего проектирования чаще всего применяется при разработке нового программного комплекса, а метод восходящего проектирования – при модификации уже существующего комплекса.

### 3. Стандарт оформления кода

**Стандарт оформления кода** – набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования.

Стандарт оформления кода (или стандарт кодирования, или стиль программирования) (англ. coding standards, coding convention или programming style).

Это набор соглашений, который принимается и используется некоторой группой разработчиков программного обеспечения для единообразного оформления совместно используемого кода.

**Целью** принятия и использования стандарта является упрощение восприятия программного кода человеком, минимизация нагрузки на память и зрение при чтении программы.

**Стандарт оформления кода описывает:**

- способы выбора названий и используемый регистр символов для имен переменных и других идентификаторов:
  - запись типа переменной в ее идентификаторе;
  - 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:**

- отсутствие магических чисел;
- ограничение размера кода по горизонтали (чтобы помещался на экране) и вертикали (чтобы весь код файла держался в памяти);
- ограничение размера функции или метода – в размер одного экрана.

## Рекомендации по стилю оформления кода в C++

### Пробелы и отступы

После зарезервированных ключевых слов языка C++ следует ставить пробел. Ставьте пробелы между операторами и операндами. Отделяйте пробелами фигурные скобки, запятые и другие специальные символы. Оставляйте пустые строки между функциями и между группами выражений.

### Именованье

Основные правила стиля кодирования приходятся на именованье. Вид имени сразу же (без поиска объявления) говорит нам что это: тип, переменная, функция, константа, макрос и т.д. Правила именования могут быть произвольными, однако важна их согласованность, и правилам нужно следовать.

#### Общие принципы именования

- используйте имена, который будут понятны даже людям из другой команды;
- имя должно говорить о цели или применении объекта;
- не экономьте на длине имени;
- не используйте аббревиатур; исключение: допускаются только известные аббревиатуры;
- не сокращайте слова.

Отметим, что типовые имена также допустимы: `i` для итератора или счётчика, `T` для параметра шаблона.

#### Примеры:

`firstName, homeworkScore`

Выбирайте подходящий тип данных для переменных. Если переменная содержит лишь целые числа, то определяйте её как `int`, а не `double`.

```
int count;           // Глобальная переменная
```

```
void func1() {        // Плохая практика
    count = 42;
}
```

```
int func1() {         // Хорошая практика!
    return 42;
}
int main() {
    int count = func1();
}
```

## *Имена файлов*

Имена файлов должны быть записаны только строчными буквами, для разделения можно использовать подчёркивание (\_) или дефис (-). Используйте тот разделитель, который используется в проекте.

## *Имена типов*

Имена пользовательских типов начинаются с прописной буквы, каждое новое слово также начинается с прописной буквы. Подчёркивания не используются:

```
MyExcitingEnum.
```

## *Имена переменных*

Имена переменных (включая параметры функций) пишутся строчными буквами, возможно с подчёркиванием между словами (в одном стиле):

```
line, lineAccount
```

! Следует инициализировать переменные в месте их объявления.

Префикс n следует использовать для представления количества объектов:

```
nPoints, nLines // Обозначение взято из математики
```

Переменным-итераторам следует давать имена

```
i, j, k // и т.д.
```

## *Имена констант*

Объекты объявляются как `const` или `constexpr`, чтобы значение не менялось в процессе выполнения. Имена констант константы пишутся в верхнем регистре. Подчёркивание может быть использовано в качестве разделителя.

```
OK, OUT_OF_MEMORY
```

Именованные константы (включая значения перечислений) должны быть записаны в верхнем регистре с нижним подчёркиванием в качестве разделителя.

## *Имена функций*

Названия методов и функций должны быть глаголами, быть записанными в смешанном регистре, начинаться с прописной буквы (в нижнем регистре) и каждое слово в имени пишется с прописной буквы:

```
getName();  
computeAverage();  
findNearestVertex();
```

## *Именованное пространство имён (namespace)*

Пространство имен называется строчными буквами.

Пространство имён верхнего уровня – это обычно название проекта.

## *Имена перечислений*

Перечисления должны именоваться либо как константы, либо как макросы:

```
enum TableErrors {  
    OK = 0,  
    OUT_OF_MEMORY = 1,  
    SURPRISE = 2  
};
```

## *Чрезмерность*

Если вы используете один и тот же код дважды или более раз, то найдите способ удалить излишний код, чтобы он не повторялся.

## *Комментарии*

**Сложный код, написанный с использованием хитрых ходов, следует не комментировать, а переписывать!**

Следует **делать** как можно меньше комментариев, делая код самодокументируемым путем выбора правильных имен и создания ясной логической структуры.

## *Эффективность*

Вызывая большую функцию и используя результат несколько раз, сохраните результат в переменной вместо того, чтобы постоянно вызывать данную функцию.

```
int index = lineAccount;    // Хорошая практика  
if (index >= 0) {  
    return index;  
}
```

```
if (lineAccount >= 0) {      // Плохая практика  
    return lineAccount;  
}
```

## Оформление

Основной отступ следует делать в два пробела

```
for (i = 0; i < nElements; i++)  
    a[i] = 0;  
....
```

```
for (initialization; condition; update) {  
    statements;  
}
```

```
while (!done) {  
    doSomething();  
    done = moreToDo();  
}
```

```
do {  
    statements;  
} while (condition);
```

## Проектирование функции

Хорошо спроектированная функция имеет следующие характеристики:

- полностью выполняет четко поставленную задачу;
- не берет на себя слишком много работы;
- не связана с другими функциями бесцельно;
- хранит данные максимально сжато;
- помогает распознать и разделить структуру программы;
- помогает избавиться от лишней работы, которая иначе присутствовала бы в программе.

## Рекомендации

При использовании операторов управления (if / else, for, while, других) всегда используйте {} и соответствующие отступы, даже если тело оператора состоит лишь из одной инструкции:

```
if (size == 0) {           // Хорошая практика  
    return;  
}
```

Возвращайте результаты проверки логического выражения (условия) напрямую:

```
return score1 == score2; // Хорошая практика
```

```
if (score1 == score2) { // Плохая практика
    return true;
} else {
    return false;
}
```

Проверка значения логического типа:

```
if (x) { // Хорошая практика
    ...
} else {
    ...
}
```

```
if (x == true) { // Плохая практика
    ...
} else if (x != true) {
    ...
}
```