1. Объекты данных и их атрибуты. Имя определяет символическое обозначение, идентификатор, по которому объекту происходит обращение в программе (необязательный

Данные - это информация, представленная в *формализованном* атрибут).

виде, позволяющим передавать хранить и обрабатывать её. *c) значение*

Слово (машинное слово) - минимальная адресуемая единица Определяет комбинацию битов, содержащихся в объекте данных.

информации (раньше было так, но сейчас минимальной адресуемой единицей это байт ).

d) *Тип данных*

Тип данных - это некоторый класс объекта данных вместе с

Объект данных - это элемент данных или несколько элементов набором операций для работы с этими объектами

данных, рассматриваемых при выполнении программы, как единое целое

Элемент данных - это отдельный фрагмент формализованной

Тип определяет:

1) класс объекта данных 2) набор операций

информации. 3) множество возможных значений и размер выделяемой области памяти

Объекты данных делятся на: Тип данных дополнительно определяет множество возможных a) *определяемые программистом (разработчикам) и* значений, которые могут содержаться в объекте данных, как следствие

*определяемые системой программирования* Объекты данных, которыми управляет программа:

объем памяти.

4) способ интерпретации типа (значения объекта данных)

● программный стек ● файловые буферы

● списки свободной памяти

Значение, то есть набор битов, в зависимости от типа может интерпретироваться по-разному, как число, как символ, как логическая переменная и так далее.

*b) элементарные (скалярные, примитивные) и структуры данных (составные, структурированные)*

Объект данных принято называть элементарным (скалярным, примитивным) если хранящееся в нем значение *всегда* фигурирует в программе как *единое целое* (нельзя получить доступ к части элементарного объекта данных, т. е. к конкретному биту).

Если объект данных может быть представлен в виде совокупности других объектов, то такие объекты называются составными или структурированными - это структуры данных.

Объединение объектов - это структуризация

*c) статические* *и* *динамические* *(динамические* *объекты данных делятся на автоматические и явные динамические)*

Объект данных характеризуется некоторым *набором свойств*, которые принято называть атрибутами.

Основные атрибуты объектов данных: *a) адрес*

Местоположение объекта данных определяет его размещение в области памяти, то есть его адрес.

*b) идентификатор*

2. Атрибуты объектов данных и понятие связывания. *3. Значение*

Если объект данных обладает определённым свойством, Определяет комбинацию битов, содержащихся в объекте данных.

выбранным из набора, определяемого тем или иным атрибутом, то говорят, что объект данных связан с этим свойством. (Набор свойств для

Связывание со значением обычно при выполнении программы (позднее связывание), но может быть выполнено и при трансляции: если

атрибута определяется при появлении языка программирования) объект из сегмента данных (глобальные и статические локальные Связывание - это выбор некоторого свойства из определённого объекты данных). Значение может меняться если это значение

набора допустимых свойств.

Период времени в течении которого выполняется связывание называется временем связывания.

Время связывания бывает

переменной и не может меняться, если это значение константы.

*4. Тип данных*

Тип данных - это некоторый класс объекта данных вместе с

*● ранее (на этапе компиляции) ● позднее (на этапе выполнения)*

набором операций для работы для работы с этими объектами Тип определяет:

С точки зрения быстродействия более эффективным является раннее связывание, но позднее связывание является более гибким.

*a) класс объекта данных b) набор операций*

c) *множество возможных значений*

Объект данных характеризуется некоторым набором свойств, Тип данных дополнительно определяет множество возможных

которые принято называть атрибутами. Основные атрибуты объекты данных:

*1. Адрес*

Местоположение объекта данных определяет его размещение в области памяти, то есть его адрес.

Местоположение обычно определяется автоматически определяет не разработчик.

значений, которые могут содержаться в объекте данных, как следствие объем памяти.

d) *способ интерпретации типа (значения объекта данных)* Значение, то есть набор битов, в зависимости от типа может

интерпретироваться по-разному, как число, как символ, как логическая переменная и так далее.

Адрес глобальных переменных или статических локальных Возможность изменения типа данных закладывается при создании определяется транслятором (или программистом, если есть такая языка программирования: статическая или динамическая типизация. возможность), то есть раннее связывание.

Для динамических переменных, переменных подпрограмм и так далее адрес определяется во время выполнения, то есть позднее связывание.

Адрес может меняться, например, при многократном вызове функций, локальные переменные каждый раз имеют новый адрес.

Не всегда элементы данных располагаются последовательно, Например, массив располагается непрерывно, а структура обычно размещается не по порядку.

*2. Идентификатор*

Имя определяет символическое обозначение, идентификатор, по которому объекту происходит обращение в программе (необязательный атрибут).

Имя определяется программистом, связывание ранее так, как производится при трансляции имя может быть множественным (адрес один, а имен несколько) и во время программы не меняется.

3. Переменные и константы. Левостороннее и 4. Время жизни объекта данных и соответствующая

правостороннее значение объекта данных.

Один из атрибутов объектов данных - значение.

Определяет комбинацию битов, содержащихся в объекте данных. Связывание со значением обычно при выполнении программы

классификация объектов данных.

Время жизни объекта данных - это время, в течении которого *можно использовать* объект данных. Для любого объекта данных это время отсчитывается от момента выделения этому объекту области

(позднее связывание), но может быть выполнено и при трансляции: если памяти и до момента освобождения памяти. Таким образом, объект из сегмента данных (глобальные и статические локальные классификация на статические и динамические объекты данных

объекты данных).

В зависимости от возможности менять свое значение в процессе выполнения программы объекты данных делятся на переменные и константы.

является классификацией по времени жизни.

По времени жизни объекты данных делятся на: *●статические*

*●динамические*

Переменная - это объект данных, связывание которого с его значением, может изменяться в течение времени жизни.

○ *автоматические (стековые)*

Пример: локальные переменные функции (размещаются в

Константа - это объект данных, связывание которого с его стеке), автоматические переменные параметры функции.

значением в течение времени жизни является неизменным. Константы бывают:

*○ явные динамические*

Если в ЯП есть механизмы, которые позволяют разработчику

*● именованные* самостоятельно выделять освобождать память.

*● буквальные (литералы)* Явные динамические переменные, это переменные, под Если в качестве имени константы используется произвольный которые программист явно выделяет память.

идентификатор, то константа называется именованной, а если в *○ неявные динамические (сомнительно)*

качестве имени константы используется одна из форм представления её значения, то константа является буквальной.

Память выделяется, когда происходят события, например инициализация.

Ссылку на местоположение объекта данных (адрес ОД) принято Статические объекты данных сохраняются до конца работы называть левосторонним значением объекта данных (L-значение, программы, а динамические могут удаляться в процессе выполнения L-value). При этом саму величину значения, содержащуюся в области программы.

хранения данных, называют правосторонним значением объекта Сегмент кода (Code Segment, CS - регистр, в котором хранится

данных (R-значение, R-value). Пример.

начало сегмента) - область памяти, которая содержит код и команды. Сегмент данных (Data Segment, DS - регистр, в котором хранится

А = С; начало сегмента) - область памяти, которая содержит статические A - левостороннее значение объекта данных, ссылка на объекты данных - глобальные переменные и константы, статические

местоположение

C - правостороннее значение объекта данных, значение, которое содержится в С

локальные переменные.

(нужен для хранения тех данных, которые не исчезают и не появляются, время жизни у таких переменных равно работе самой программы - статические);

То, что хранится в сегменте данных часто инициализируется 0, но не всегда.

Сегмент стека (Stack Segment, SS - регистр, в котором хранится

начало сегмента) - содержит автоматические стековые переменные, локальные параметры функции, если не объявлены, как статические.

(специальная область, там размещаются те переменные, которые автоматически появляются в процессе работы программы (локальные переменные функции) (без инициализации переменные содержат мусор, то есть произвольное значение, хранившееся в данном участке памяти);

Куча (Heap) - содержит явные динамические объекты данных. Выделением памяти, освобождением памяти и ведением списка

свободных адресов занимается монитор кучи.

(если куча пустая, то выделяется первый блок, сколько нам нужно,

5. Область действия и область видимости объектов данных. Соответствующая классификация объектов данных.

Область действия объекта данных - это область программного кода, где может использоваться конкретный объект данных.

потом если мы отдали эту память, то в следующий раз отдается в По области действия объекты данных делятся на

первую очередь эта память); ● *глобальные* - можно использовать в любом месте программного кода

● *локальные* - можно использовать только в части кода Область видимости объекта данных - это часть область действия,

где возможен обычный доступ к объекту данных.

Локальную и глобальную переменные можно объявлять с одним и тем же именем. При этом *объявление имени переменной в блоке скрывает объявление этого же имени во внешнем блоке или глобальное имя переменной*. После выхода из блока имя внешней по отношению к этому блоку переменной восстанавливает свой смысл.

Пример:

Если в подпрограмме есть переменная, имя которой совпадает с именем внешней переменной, то внешняя переменная “не видна” в подпрограмме. Для обращения к внешней переменной, необходимо использовать операцию разрешения области видимости (::).

Область видимости совпадает с областью действия за исключением ситуации, когда во вложенном блоке описана переменная с таким же именем. В этом случае внешняя переменная во вложенном блоке невидима, хотя он и входит в ее область действия. Тем не менее к этой переменной, если она глобальная, можно обратиться, используя операцию доступа к области видимости ::. Способ обратиться к скрытой локальной переменной отсутствует.

Пример в конспекте

6. !! Понятие пространства имен. Смысл и особенности *1) Объявление пространства имен*

использования.

Пространство имён (namespaces) - это область логического группирования уникальных идентификаторов. (или по-другому, — это механизм в программировании, который позволяет организовывать код в логические группы, предотвращая конфликты имен и улучшая структуру проекта.)

:: - операция разрешения области видимости + пример в конспекте

Пространство имён определяется блоком инструкций: **namespace foo** {

int bar; }

Внутри этого блока идентификаторы могут вызываться именно так, как они были объявлены. Но вне блока требуется указание имени пространства имён перед идентификатором. Например, вне namespace foo идентификатор bar должен указываться как foo::bar.

*2) Использование пространства имен*

Смысл использования:

● *Предотвращение конфликтов имен:* Пространства имен позволяют изолировать код, гарантируя, что имена функций, классов и переменных, объявленных в одном пространстве имен, не будут конфликтовать с именами в других пространствах имен.

● *Организация кода:* Пространства имен помогают структурировать

C++ содержит некоторые другие конструкции, делающие подобные требования необязательными. Так, при добавлении строки

**using namespace foo**;

в код, указывать префикс foo::больше не требуется. Ещё пример: **namespace Namespace12** {

int foo=0; }

проект, разделяя его на логические блоки, связанные общей функциональностью.

● *Повторное* *использование* *кода:* Код, организованный в пространства имен, легче использовать повторно в других проектах, не опасаясь конфликтов имен.

void func1() {

**using namespace Namespace12**;

*// теперь все имена из Namespace12 будут видны здесь без дополнительных префиксов*

++foo;

Идентификатор, определённый в пространстве имён, }

ассоциируется с этим пространством. Один и тот же идентификатор может быть независимо определён в нескольких пространствах.

Пространства имен используются для организации кода в виде логических групп и с целью избежания конфликтов имен, которые могут возникнуть, особенно в таких случаях, когда база кода включает несколько библиотек.

Все идентификаторы в пределах пространства имен доступны друг другу без уточнения. Идентификаторы за пределами пространства имен могут получить доступ к членам с помощью полного имени для каждого идентификатора, например, с помощью объявления для одного идентификатора (using std::string) или директивы using для всех идентификаторов в пространстве имен (using namespace std;).std::vector<std::string> vec; Код в файлах заголовков всегда должен содержать полное имя в пространстве имен.

Пространства имен могут быть вложенными друг в друга для создания более детальной иерархической структуры.

void func2() {

*// а тут имя нужно уточнить:* Namespace12::foo = 42;

}

*3) Зачастую пространства имён в C++ используются для избежания коллизий имён*

**namespace** { int a;

void f() { */\*...\*/* } int g() { */\*...\*/* }

}

*4) Также можно сделать видимым не всё пространство, а отдельные имена внутри него, например:*

**namespace foo** { int bar;

int somelse; }

int main () {

**using** foo::bar; *//Делает видимым только bar,*

*somelse невидим!* Особенности использования см. на след. странице **return** 0;

}

7. Понятие типа данных в языках программирования. Смысл и особенности использования.

Тип данных - это некоторый класс объекта данных вместе с набором операций для работы для работы с этими объектами

В каждом языке программирования имеется некоторый набор встроенных элементарных типов данных. Дополнительно язык может предоставить возможности, позволяющие программисту определять новые типы данных.

Тип определяет:

a) *класс объекта данных b) набор операций*

*c) множество возможных значений*

Тип данных дополнительно определяет множество возможных значений, которые могут содержаться в объекте данных, как следствие объем памяти.

*d) способ интерпретации типа (значения объекта данных)* Значение, то есть набор битов, в зависимости от типа может

интерпретироваться по-разному, как число, как символ, как логическая переменная и так далее.

Возможность изменения типа данных закладывается при создании языка программирования: статическая или динамическая типизация.

Тип данных может быть задан явно и неявно.

Преимущества от использования типов данных

Типы данных защищают программы по крайней мере от следующих ошибок:

1. Некорректное присваивание. Пусть переменная объявлена как имеющая числовой тип. Тогда попытка присвоить ей символьное или какое-либо другое значение приведет к ошибке еще на этапе компиляции. Такого рода ошибки трудно отследить обычными средствами.

2. Некорректная операция. Типизация позволяет избежать попыток применения выражений вида “Hello world” + 1. Поскольку, как уже говорилось, все переменные в памяти хранятся как наборы битов, то при отсутствии типов подобная операция была выполнима (и могла дать результат вроде “Hello world”!). С использованием типов такие ошибки отсекаются опять же на этапе компиляции.

3. Некорректная передача параметров в процедуры и функции . Если функция “синус” ожидает, что ей будет передан числовой аргумент, то передача ей в качестве параметра строки “Hello world” может иметь непредсказуемые последствия. При помощи контроля типов такие ошибки также отсекаются на этапе компиляции или приводят к ошибкам выполнения программы, если значения параметра вводятся с клавиатуры или файла.

Кроме того, типы данных позволяют программисту абстрагироваться от машинного представления информации в виде наборов нулей и единиц и строить программы, основываясь на знакомых понятиях, таких, как числа, множества, последовательности, и т.п. В конечном итоге это приводит к получению более надежных программ.

Текущие значения атрибутов объекта данных хранятся в дескрипторе (описателе) данных. Дескриптор может быть частью объекта данных или храниться в отдельной области памяти.

Под реализацией типа данных понимают

1) Способ представления объектов данных этого типа в памяти компьютера во время выполнения программы;

2) Способ представления операций, определенных для этого типа данных (комбинация аппаратных и программных средств, реализующих конкретные алгоритмы и процедуры над представлениями объектов данных заданного типа в памяти).

Смысл и особенности использования типов данных:

1. Организация данных: Типы данных позволяют структурировать информацию, делая код более понятным и удобным для работы.

2. Предотвращение ошибок: Компилятор или интерпретатор может использовать информацию о типах для выявления ошибок на ранних этапах разработки (например, попытки сложить число и строку).

3. Оптимизация: Знание типов данных позволяет компилятору генерировать более эффективный машинный код, оптимизированный под конкретные операции.

4. Абстракция: Типы данных позволяют абстрагироваться от деталей реализации и работать с данными на более высоком уровне.

8. Понятие контроля типов в языках программирования. из-за ошибок, связанных с типами. Она обеспечивает выполнение Назначение и особенности. операций только над совместимыми типами и предотвращает

Контроль типов означает что к каждой функции на вход приходит правильное число аргументов правильного типа.

*Статический контроль* типов не нужен во время выполнения, а нужен только во время компиляции.

Если при помощи контроля типа в языке программирования принципиально возможно обнаружить все без исключения ошибки такого контроля, то ти , а язык программирования является

Если это не выполняет называется слабо типизированным.

потенциальные ошибки памяти или уязвимости безопасности, которые могут возникнуть из-за несоответствия типов.

Языки программирования со строгим контролем типов имеют ряд преимуществ.

• В отсутствие контроля типов работа программ в большинстве языков может завершиться *непредсказуемо*.

•Цикл редактирование–компиляция–отладка трудоемок, поэтому *раннее обнаружение ошибок* крайне желательно.

• Объявление типов *облегчает документирование* программ.

Основным понятием контроля типов является *соответствие* • *Более эффективный объектный код*, если типы объявлены явно.

*(conformance)*. Рассмотрим, например, абстракцию физической единицы измерения. Если разделить расстояние на время, мы получим число, означающее скорость, а не вес. Аналогично, деление единицы силы на температуру бессмысленно, а деление единицы силы на массу — нет. Эти примеры относятся к строгому контролю типов, поскольку *правила исследуемой предметной области четко определены и регламентируют допустимые сочетания абстракций*.

Назначение контроля типов - повысить надежность, безопасность и корректность программного обеспечения путем выявления ошибок, связанных с типами, во время компиляции или выполнения программы.

Языки, в которых контроль типов отсутствует, обладают *большей гибкостью*, но и в этом случае “программисты практически всегда знают, какие объекты ожидаются в качестве аргументов сообщения и какие подлежат возвращению”. На практике *безопасность*, обеспечиваемая языками программирования со строгим контролем типов, *компенсирует потерю гибкости*, присущей языкам без контроля типов, особенно при проектировании *крупномасштабных* систем.

Обеспечивая контроль типов, языки программирования могут помочь обнаружить и предотвратить распространенные ошибки программирования, такие как передача аргумента несовместимого типа в функцию или выполнение операций над несовместимыми типами данных.

Контроль типов в языках программирования обычно включает в себя следующие функции:

1. Проверка типов: Процесс проверки типов переменных, выражений и параметров функций в программе. Это может быть сделано статически (во время компиляции) или динамически (во время выполнения).

2. Некоторые языки программирования используют значение для автоматического определения типов переменных и выражений на основе их использования в программе. Это позволяет программистам писать код без явного указания типов, уменьшая необходимость в повторяющихся аннотациях типов.

3. Сильная типизация: Языки программирования с сильной типизацией обеспечивают строгий контроль типов, гарантируя, что несовместимые операции или присваивания недопустимы. Языки с сильной типизацией обычно требуют явного преобразования типов или приведения при выполнении операций над различными типами.

4. Безопасность типов: Безопасность типов гарантирует, что программа не будет выполнять неожиданные или опасные действия

9. Статическая и динамическая типизация в языках Именно действия и операции над данными предоставляют информацию

программирования.

Возможность изменения типа данных закладывается при создании языка программирования.

Типизация бывает:

о типе).

Дескриптор типа хранит информацию о типе объекта данных в данный момент времени.

Может быть, реализован только динамический контроль типов.

*● статическая ● динамическая*

Плюсы динамической типизации: 1) *гибкость* программирования

Статическая типизация означает, что *связывание* объекта данных с типом является *ранним*, то есть выполняется *на этапе компиляции*, а не на этапе выполнения, и *не меняется* в процессе выполнения.

При статической типизации, не нужен дескриптор типа (блок или “довесок” содержащий информацию о том, какого типа объект данных в данный момент).

2) хорошо реализуется при помощи *интерпретатора* 3) более *простой синтаксис* и меньше многословие

4) *облегчение работы с внешней средой* (если мы не знаем какие данные нам поступят, то ничего страшного, такой и будет у них тип)

Минусы динамической типизации

Плюсы статической типизации 1) *понижение надёжности* контроля типов, следовательно, 1) отсутствие необходимости контролировать типы понижение безопасности кода

динамически. *Контроль типов* может быть осуществлен *на этапе компиляции*

*2) отсутствие дескрипторов типа*

1)+ 2) приводят к простоте реализации, что дает преимущества 3) и 4)

3) существенный *выигрыш в скорости вычисления* на этапе выполнения, в том числе из-за упрощения машинного кода. Машинный код компактнее и проще

4) *экономия памяти* и её эффективное использование.

Значит статическая типизация хорошо подходит для сложного, но быстрого кода.

2) существенное *снижение скорости выполнения*

3) *накладные расходы памяти* на дескриптор типа и других структур.

Минусы статической типизации:

1) меньшая гибкость (нельзя менять по ходу программы);

2) под интерпретатор это ложится не очень хорошо; плохая реализация при помощи интерпретатора.

3) проблема взаимодействия с внешней средой (Например, при работе с базами данных, когда мы не знаем какого типа данные поступают извне)

Динамическая типизация означает, что *связывание* с типом данных происходит *на этапе выполнения*, то есть является *поздним*, и *может меняться* в процессе выполнения программы.

Информация о типах достается из (операций) действий над ними, например присваивания.(Вывод типов (Type Inference) - это способность компилятора или интерпретатора *автоматически определять тип данных* переменной *на основе контекста*, в котором она используется.

10. Сильная и слабая типизация в языках программирования. Не настолько жестко фиксирует правила. *Действия для одного* Контроль типов означает что к каждой функции на вход приходит *типа можно выполнять по отношению к другим* — правда, с

правильное число аргументов правильного типа.

Статический контроль типов не нужен во время выполнения, а нужен только во время компиляции.

Если при помощи контроля типа в языке программирования

непредсказуемым результатом. Например, можно сложить строку и число.

Преимущества:

принципиально возможно обнаружить все без исключения ошибки такого контроля, то ти , а язык программирования является

Если это не выполняет называется слабо типизированным.

1. Сильная типизация:

Определение: Языки с сильной типизацией *строго контролируют*

● Большая *гибкость*: Легко преобразовывать данные между разными типами.

● *Удобство* работы со смешанными типами данных. Например, целочисленные и вещественные числа.

● *Меньше кода*: Не требуется явное приведение типов.

Недостатки:

*совместимость типов данных* и *не допускают неявных преобразований*, ● *Повышенный риск ошибок*: Неявные преобразования могут которые могут привести к потере информации или непредсказуемому привести к неожиданному поведению. Требует большего внимания поведению. Разработчикам при создании программного кода со стороны разработчика: в некоторых ситуациях типы могут

потребуется придерживаться *конкретных принципов работы с видами информации*.

Это *жестко прописанные правила работы с каким-то типом*.

вести себя непредсказуемо.

● Результат на экране может оказаться совершенно не таким, каким его хочет видеть разработчик.

Если переменная в строго типизированном языке числовая, значит, с ● *Снижение читаемости* кода: Не всегда очевидно, какие

ней можно выполнять только действия, предназначенные для чисел. Например, математические операции с числами допустимы, а вот если попытаться применить их к строкам — программа выдаст ошибку.

преобразования происходят.

Примеры языков: JavaScript, PHP, Perl.

Преимущества:

● Повышение *надежности*: Меньше ошибок, связанных с несовместимостью типов.

● Улучшение *читаемости* кода: Типы данных явно указаны, что делает код более понятным.

● Лучшая *оптимизация*: Компиляторы могут генерировать более эффективный код, зная типы данных.

*Недостатки:*

● *Менее гибкий код*: Требует большего количества кода для явного приведения типов, если это необходимо

Примеры языков: Java, C#, Python, Go.

*2. Слабая типизация:*

Определение: Языки со слабой типизацией более гибкие в отношении типов данных и позволяют выполнять *много неявных преобразований*.

11. !!! Структуры данных. Преимущества применения и Организация - это компоненты и связи между ними.

возможные варианты классификации.

Структура данных - это *объект данных*, который может быть

Структура *фиксированного размера как динамический объект* данных. Примеры: динамический массив, структура типа Struct в

представлен в виде *совокупности* некоторых *других данных* или качестве локальной переменной.

компонент при этом каждая компонента, может быть как *Динамическая структура* данных с переменным размером *в элементарным* объектом данных, так и *другой структурой данных*. *качестве статической переменной*. Пример: файл, объявленный

*Процесс объединения* объектов данных *в единую структуру* глобально. данных называется структуризацией.

*Способ объединения* данных в структуру данных, называется 2) Тип компонент структуры

механизмом структуризации.

Структуры данных, как и элементарные объекты данных, по

Однородная (гомогенная) - включает в свой состав компоненты только одного типа (*однотипные элементы*). Например, массив или

времени жизни могут быть *статическими* и *динамическими*. файл.

Динамические делятся на *автоматические* и *явные динамические*. Неоднородная (гетерогенная) - объединяет в своём составе Также у структур данных есть *область видимости*, о*бласть компоненты разных типов.* Например, структура типа Struct.

*действия*, и они могут *относиться* к некоторому *пространству имён*.

По области действия структуры данных делятся на: 3) Механизм доступа к компонентам структуры (механизм

● *глобальные ● локальные*

выборки)

Прямой (произвольный) означает, что любая компонента

*● внешние по отношению к чему-либо* структуры *всегда непосредственно доступна* с помощью некоторых средств. Например доступ к элементам массива по индексу или к полям Struct по имени.

Атрибуты структур данных: Последовательный означает, что *доступ в данный момент есть*

*● тип*

*● местоположение ● имя*

*● значение*

Особенности структурированных типов данных.

*только к одному элементу*. Например, традиционный файл или стек. *Для доступа к другим компонентам необходима обработка предыдущих компонент*, например перебор или прочтение строк.

ПРЕИМУЩЕСТВА !? скорее всего что нибудь про гибкость и более точное представление данных предметной области, повышение уровня

Дополнительные атрибуты, структур данных или *способы классификации*.

абстракции и т.д. + про то что концепция данных богаче + возможно структура лучше показывает связь между компонентами

1) Число компонент структур.

Фиксированный размер структуры: число компонент *не меняется*

С инета:

Структуры данных играют решающую роль в разработке

в течение времени жизни (жёсткая структура). программного обеспечения по нескольким причинам: Пример: массив, структуры типа Struct

Переменный размер: число компонент *меняется* в течении её 1. Эффективность. Правильно выбранные структуры данных времени жизни. оптимизируют доступ и обработку данных, что приводит к созданию Пример: файл, стек, очередь, список. более быстрого и эффективного программного обеспечения. Правильная структура данных может значительно сократить

Структура также может иметь *внутренние* *связи* *между* временную сложность стандартных операций, таких как поиск, *компонентами*, которые могут меняться с течением времени жизни. вставка и удаление.

Динамическая структура данных - это структура данных, у 2. Масштабируемость. По мере роста приложений и обработки больших

которой в течении времени жизни могут *изменяться число компонент или связи*, то есть *меняется организация* структуры данных.

объемов данных эффективные структуры данных становятся еще более важными. Хорошо спроектированная структура данных может

поддерживать расширение приложений без серьезного снижения 12. Типовые средства языка программирования для производительности, гарантируя, что программное обеспечение реализации концепции связанного динамического распределения

остается отзывчивым и стабильным.

3. Сопровождение кода. Приложение с организованными структурами

памяти. Указатели.

*Стандартные структуры* данных *не позволяют учитывать связи*

данных легче поддерживать, изменять и расширять. Выбор между компонентами, которые существуют в предметной области, подходящих структур данных упрощает код и способствует Следовательно, *неадекватное отражение реальности*.

улучшению практики кодирования, тем самым улучшая качество 1) *Состав* реальных информационных *структур может* программного обеспечения. *меняться*, причём достаточно сильно и достаточно быстро.

4. Разработка алгоритмов. Поскольку большинство алгоритмов 2) *Внутренняя* *организация* информации, может также

построены на основе одной или нескольких структур данных, их эффективность во многом зависит от базовых структур. Правильная

*меняться*: могут появляться и исчезать дополнительные структуры

структура данных позволяет лучше реализовать алгоритм и может 3) В реальности информационное пространство имеет *большое*

существенно повлиять на производительность программного обеспечения.

*количество связей*. Связи, могут иметь *различный смысл* и вид.

Для того, чтобы наиболее *гибко и адекватно отражать* информационные структуры предметной области, были введены *динамические структуры*.

Основная идея механизма связанного распределения памяти - это введение *специальных компонент* в элементы структуры данных, которые будут *определять относительное расположение элементов*.

Этот компонент указатель (ссылка), который ссылается (указывает) на соседние элементы. При этом в памяти элементы могут располагаться как угодно.

Для реализации связанного динамического распределение памяти необходимо поддержать

1) *область памяти* под динамические структуры (под явные динамические объекты) - куча (Heap).

*2) Средство распределения и организации памяти кучи*

a) в процессе работы программы динамически распределять (*выделять*) память под обобщённые элементы данных

b) средство *освобождения* памяти неиспользуемых элементов данных.

Это происходит с помощью монитора (администратора кучи) и подпрограмм языка программирования (возможно, имеются ввиду средства явного распределения памяти).

3) *Механизмы связи элементов данных друг с другом*: указатели и операции над ними. Следовательно, нужны специальные типы данных, ссылочные и адресные.

Указатель - это *специальный тип данных* который содержит *сведения о местоположении* других объектов данных.

Указатели бывают *типизированные и нетипизированные*.

Типизированные - ссылочный тип. Указатель ссылается на 13. Указатели в языках программирования. Назначение и *конкретный тип данных*, он *знает тип*, на который он ссылается. Тип, особенности использования.

на который он ссылается - базовый тип. Указатель на объект данных. Указатель - это специальный тип данных который содержит Нетипизированные - адресный тип (нейтральные). *Не знает*, на сведения о местоположении других объектов данных.

какой тип адресуется. Указатель на *void* - void\*.

Указатель понимается как *адрес памяти как таковой*, с 1) Указатели бывают *типизированные и нетипизированные*. неопределенной организацией и неизвестной размерностью указуемой Типизированные - ссылочный тип. Указатель ссылается на переменной. Его *можно* *присваивать,* *передавать* *в* *качестве* конкретный тип данных, он знает тип, на который он ссылается. Тип, на

*параметра и результата функции,* менять тип указателя, но *операции косвенного обращения и адресной арифметики с ним недопустимы.*

НЕ ответ, а просто пояснение для себя:

Дословно выходит, что это указатель на пустоту, но реально - это указатель на элемент любого типа. Соответственно, указателю void\* можно присвоить значение любого указателя.

Например, если у нас есть две простых переменных и два указателя:

|  |  |  |
| --- | --- | --- |
| int i = 3; double x = int\* pi =  double\* pd |  | |
| 1. &i;  = | 44; |
| &x; |

и есть нетипизированный указатель p, то можно сделать все следующие присваивания:

|  |  |
| --- | --- |
| void\* p; //указатель на любые данные p = pi; //p указывает туда же, что и p = pd; //p указывает туда же, что и p = &i; //p хранит адрес переменной  p = &x; //p хранит адрес переменной |  |
| pi pd i  x |

и даже такие:

случае, в pпомещаем адрес, который содержится в указателе pi, если учитывать инициализацию переменной piвыше по коду, то это адрес переменной i, а во втором адрес самого указателя - т.е. адрес, по которому хранится переменная pi.

Зачем же используется нетипизированные указатели? И в языке C++ и, тем более, в языке Си данный вид указателей может использоваться для низкоуровневых операций, когда не важно, что лежит по некоторому адресу. Примером может служить функция fwrite, которая записывает некоторый участок в памяти в файл. В первом параметре задаётся указатель на некоторый участок памяти (т.е. адрес начала этого участка) и так как для неё не важно, что лежит в этой памяти, передается указатель типа void\*, или точнее **const void**\*- так как мы не хотим менять память на данному указателю. Подробнее про использование слова const с указателями смотри на следующем шаге.

Другой похожий пример - похожие функции m e m c py и m e m m ove . Это

низкоуровневые функции копирования (и перемещения) участка памяти языка Си.

который он ссылается - базовый тип. Указатель на объект данных. Нетипизированные - адресный тип (нейтральные). Не знает, на

какой тип адресуется. Указатель на void - void\*.

Указатель на void понимается как *адрес памяти как таковой*, с неопределенной организацией и неизвестной размерностью указуемой переменной. Его *можно* *присваивать,* *передавать* *в* *качестве параметра и результата функции,* менять тип указателя, но *операции косвенного обращения и адресной арифметики с ним недопустимы.*

*2) Пустая ссылка*

Пустая ссылка, пустой адрес, адресный нуль, нулевая ссылка, нулевой адрес (сомнительно, ему это не нравится) - специальное зарезервированное ключевое слово (константа), обозначающее то, что ссылка ни на что не ссылается.

В С - *NULL*, с C++ 11 - *null\_ptr*

a) NULL и null\_ptr *совместимы с указателями любого типа* b) *Можно проверить равенство* адреса с null\_ptr

+ см в конспекте

Особенности использования: ● Опасность ошибок

○ висячие ссылки - ссылки, которые ссылаются на уже освобожденный участок памяти

○ мусор - область памяти помеченная, как занятая, но на нее не указывает никакой указатель, следовательно потеря контроля над данным участком памяти и утечка памяти

● Обращение к неверной области памяти: Неправильное использование указателей может привести к чтению или записи данных в не предназначенные для этого области памяти, что вызовет непредсказуемое поведение программы или ее крах.

● Утечки памяти: Если не освобождать память, на которую указывает указатель, после того как она больше не нужна, это приведет к утечке памяти.

Синтаксис: В разных языках программирования синтаксис работы с указателями может отличаться. Например, в C/C++ для объявления

указателя используется символ `\*`, а в Java и Python работа с памятью 14. Базовые средства поддержки работы с указателями в языках

скрыта от программиста.

Указатели *позволяют работать с динамической памятью*, которая выделяется во время выполнения программы.

программирования.

|  |  |
| --- | --- |
| 1) Операции с указателями. | |
| a) | Присваивание |

Соответственно, основная операция для указателя - это косвенное обращение по нему к той переменной, адрес которой он содержит. В Си

Висячая ссылка (dangling reference) — это ссылка на область памяти, в которой нет "живого" объекта. Это возможно когда *время*

имеется специальная операция \* - (разыменование/ разадресация) *жизни ссылки дольше чем время жизни объекта*, на который она звездочка, которую называют косвенным обращением по указателю. В указывает. Ссылка становится висячей в тот момент, когда *компилятор* более широком смысле ее следует понимать как переход от *разрушает объект* (вызывает деструктор объекта и потом освобождает

переменной-указателя к той переменной (объекту), на которую он ссылается.

память, которая объектом занималась), а *ссылку ещё нет*.

Можем потерять доступ к данным и получить мусор (garbage). Может появится, если время жизни объекта данных больше, чем время жизни указателя на этот объект данных.

b) сравнение

i) (обязательно) *на равенство, неравенство* ii) (дополнительно) *на меньше, больше*

c)

d) (обязательно)

Оператор разыменования `\*` позволяет получить доступ к значению по адресу, хранящемуся в указателе.

Невозможно разыменовать указатель на void. Неопределённое поведение при разыменовании не инициализированного указателя.

e) (дополнительно) арифметические операции с указателями

i) Операция *сложения адреса с константой* и *вычитание константы из адреса*.

Смещение указателя происходит на величину *константа\* (объём памяти*, выделенный под тип данных, на который указывает указатель)

ii) Операция сложения указателей НЕ поддерживается iii) Операция *вычитания указателей*

В результате получаем число, равное разнице указателей, то есть *объём памяти, выделенный под тип данных*, то есть размер типа.

2) Операции распределения (выделения) памяти C++ - new/ C - malloc

int\* ptr = new int;

int\* ptr = new int(123);

3) Операции освобождения памяти

C++ - delete/ C - free + все примеры в конспекте

+ Ошибки, которые могут возникать + Для ЯП C

Для С++ для malloc() требуется приведение типов Нельзя смешивать функции new - delete и malloc - free

15.Указатели на функции. Назначение и особенности использования.

Особенности указателей на функцию:

1) В отличие от обычных указателей, указатель на функцию

Ответ (код) - см в конспекте

Вместо того, чтобы жестко прописывать определенное действие

указывает на код, а не на данные. Обычно указатель на функцию хранит начало исполняемого кода.

внутри алгоритма, вы передаете указатель на функцию, которая будет 2) В отличие от обычных указателей, мы не освобождаем выполнять это действие. Таким образом, *один и тот же алгоритм* память с помощью указателей на функции.

*может работать с разными функциями, изменяя свое поведение*. Указатель на функцию (function pointer) хранит адрес функции. По

3) Имя функции также можно использовать для получения адреса функции.

сути указатель на функцию содержит адрес первого байта в памяти, по 4) Как и обычные указатели, мы можно создать массив

которому располагается выполняемый код функции.

Самым распространенным указателем на функцию является ее имя (как и у массивов). С помощью имени функции можно вызывать ее и получать результат ее работы.

Но также указатель на функцию мы можем определять в виде отдельной переменной с помощью следующего синтаксиса:

указателей на функции.

5) Как и обычные указатели данных, указатель на функцию может передаваться в качестве аргумента, а также может быть возвращен из функции.

<тип возвращаемого значения функции> (\*<идентификатор указателя>)([<тип параметра 1>, <тип параметра 2>]);

*Указатель может указывать только на такую функцию, которая имеет тот же возвращаемый тип и типы параметров, что и определение указателя на функцию.*

В С++ *нет совместимости присваивания с другими указателями* В C есть совместимость и с типизированными, и с

нетипизированными указателями

При определении указателя стоит обратить внимание на скобки вокруг имени. Так, использованное выше определение: void (\*message) ();

НЕ будет аналогично следующему определению: void \*message (); Во втором случае определен не указатель на функцию, а прототип

функции message, которая возвращает указатель типа void\*.

Кроме одиночных указателей на функции мы можем определять их массивы. Для этого используется следующий формальный синтаксис:

тип (\*имя\_массива[размер]) (параметры)

*Массив указателей содержит n функций, которые могут последовательно вызыватся в цикле через перебор массива (через for).*

+ он часто говорил что это можно использовать в реализации меню

16. Функции с переменным числом параметров как пример реализации механизма работы с указателями.

Функции с переменным числом параметров (не с параметрами по

может быть корректно использовано когда все параметры передаваемые в функцию принадлежат одному и тому же типу (имеются ввиду те параметры, которые идут туда где расположено многоточие).

Даже если типы передаваемых параметров будут различны, то

умолчанию, а с переменным числом параметров) - вариативные функци.

компилятор не выдаст ошибки, но ошибки не выведется только потому что *если компилятор видит эти три точки на месте параметров, то*

Небезопасный механизм, не рекомендован к использованию. Причины небезопасности

1) *Особенности хранения* параметров функции в программном стеке

2) *Отсутствие надлежащего контроля типов* при компиляции

*он отключает проверку типов*. На самом деле легко увидеть эффект небезопасности работы с функцией.

3) Низкоуровневый механизм, сильно завязанный на *особенностях реализации*.

Расположение параметров функции в стеке:

1 способ - 1ый параметр *ближе к дну* - Pascal - “слева-направо” 2 способ - 1ый параметр *на вершине* С/С+ + - cdecl.

Возвращает контекст, зачищает за собой: ● В Pascal - *вызываемая* функция

● В С/С+ + *вызывающая* функция

Функция с переменным числом параметров должна иметь хотя бы один явный параметр.

Для любых аргументов используется повышение (продвижение) типа.

● Любых похожих на *целочисленный - до int*

● Любых похожих на *вещественные - до double*

Последний из явных параметров не должен быть повышаемым, то есть должен быть либо int и либо double.

Последний явно объявленный параметр не должен передаваться по ссылке.

Пример в конспекте ( в том числе про va\_list, va\_start, va\_arg, va\_end)

З а м еч а ние :

● В функцию требуется писать тот тип, который указали. Дополнительно (вне конспекта):

Язык программирования Си допускает использование функций, которые имеют *нефиксированное количество параметров*.

Для определения параметров неопределенной длины в таких функциях используется многоточие:

тип имя\_функции(обязательные параметры, ...)

Весь этот механизм удобно использовать тогда, когда *требуется обработка однотипных элементов*. Переменное число параметров

17. Понятие структурного подхода к разработке 2. Отсутствие внятных идей в области организации структур программного обеспечения, предпосылки возникновения и программ (нет канонов, каждый делал так, как считал преимущества применения. нужным)

Методология структурного программирования появилась как Надо было выстраивать определенную методологию (с некотрой

следствие *возрастания сложности решаемых на компьютерах задач*, и соответственно, усложнения программного обеспечения. В 1970-е годы

философией). Эта методология получила название структурного подхода. Основоположники структурного подхода:

объёмы и сложность программ достигли такого уровня, что ● Э. В. Дейкстра традиционная *(неструктурированная) разработка программ перестала* ● Ч. Э. Р.Хоар *удовлетворять потребностям практики*. Программы становились ● О. И. Дел

слишком сложными, чтобы их можно было нормально сопровождать. Принято считать, что структурный подход включает в себя Поэтому потребовалась систематизация процесса разработки и несколько идей концепций

структуры программ. *1) структурное программирование*

*2) модульное* *программирование* *(процедурное* Из конспекта: *программирование)*

В 60 годы 20 века увеличилось количество крупных проектов. *3) нисходящее проектирование + нисходящее тестирование* Задачи становились более сложными, а ПО более крупным. В 4) Бригадная организация труда - устарело

разрабатываемом коде была сумбурность и не структурированность. Среди разработчиков не было единого стиля, было трудно организовать разработку ПО в команде.

Достоинства структурного программирования по сравнению с интуитивным неструктурным программированием:

В связи с этой необходимостью появилась совокупность строго ● обоснованных (логически и математически) практик для *повышения*

*эффективности* программирования. ●

Основной идеолог структурного программирования (начало 70-х

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| уменьшение трудностей тестир | | | ования программ,  программ | как  , как |
| следствие | повышается надежность | |
| повышение производительности т | | | руда программистов стоимость програм |
| следствие | | уменьшается время и | мной |
| разработки; | | |

годов 20 века) - *Э. В. Дейкстра* Структурирование программы позволяет ле*гко находить и* 1) необходимо было ввести организацию программного кода *корректировать ошибки*, а отдельные п*одпрограммы можно*

(Программы уже в (60е годы 20 века) имели достаточно *переделывать (модифицировать) независимо* от других;

большое количество операторов при отсутствии их ● повышение ясности и читабельности программ, что нормальной организации. Это серьезно затрудняло упрощает их сопровождение;

понимание программ и согласование частей друг с другом) Следование принципам структурного программирования сделало 2) активно и *неоправданно* *использовались* *операторы* тексты программ, даже довольно крупных, *нормально читаемыми*.

*безусловного перехода go to (идет из ассемблерных программ)* Серьёзно облегчилось понимание программ, появилась возможность Многочисленные передачи управления *затрудняют понимание* разработки программ в нормальном промышленном режиме, когда программного кода, так как *последовательность выполнения* программу может без особых затруднений понять не только её автор, но операторов *не соответствует последовательности их появления* в коде и другие программисты. Это позволило разрабатывать достаточно

такая программа называется “программа-спагетти”.

Причины не структурированности кода:

1. Легкость и *простота использования оператора безусловного перехода* goto. Из-за этого оператор безусловного перехода применялся часто и часто это было неоправданно

крупные для того времени программные комплексы силами коллективов разработчиков, и сопровождать эти комплексы в течение многих лет, даже в условиях неизбежных изменений в составе персонала.

Структурное программирование позволяет значительно сократить число вариантов построения программы по одной и той же спецификации, что значительно снижает сложность программы и, что ещё важнее, облегчает понимание её другими разработчиками.

В структурированных программах *логически связанные операторы* 18. Структурная теорема как основа структурного

*находятся визуально ближе*, а *слабо связанные — дальше*, что позволяет обходиться без блок-схем и других графических форм изображения

программирования.

Подробно - см в конспекте.

алгоритмов (по сути, *сама* *программа* *является* *собственной* Теоретические основы структурного программирования *блок-схемой*). разработали итальянские математики:

● повышение эффективности объектного кода программ как с точки зрения времени их выполнения, так и с точки зрения необходимых затрат памяти.

*● К. Бош*

● *Дж. Джекобини*

(теорему о струк

Любая программа может быть построена и*з функциональных блоков*, то есть частей выполняющих определенные функции, имеющих единственный вход и единственный выход с помощью некоторых управляющих конструкций, образующих базисное множество

управляющих конструкций (минимальный размер 2 конструкции), в частности базисное множество управляющих конструкций может состоять из 3 конструкций:

1) следование

Это происходит *прозрачно от программиста*, то есть блоки выполняются *в порядке появления в программе*.

2) двоичное “бинарное” решение

*Выбор одного из двух функциональных блоков* в зависимости от *результатов некоторого теста.*

3) цикл с предусловием

*Многократное* выполнение определенного функционального блока с принятием решения о *необходимости очередного повторения* в зависимости *от результатов некоторого теста* проводимого каждый раз *перед выполнением* этого блока.

Используется неявные безусловный переход.

Блок-схема - это ориентированный граф, отражающий последовательность действий алгоритмов.

Минимальный базис состоит из двух конструкций (Доказательство - в конспекте)

1. Следование

2. Цикл с предусловием

Функциональная эквивалентность программ – при одинаковых данных на входе выходят одинаковые данные на выход.

Структурная теорема говорит *о функциональной эквивалентности* любой программы, построенной любым образом программе из базисных конструкций.

19. Структурная теорема и обоснование ее справедливости с помощью преобразования Ашкрофта Манны.

Весь 18 вопрос +

|  |  |
| --- | --- |
|  | Идеи доказательства структурной теоремы с помощью |
| преобразования Ашкрофта-Манны | |

1) Все *функциональные блоки алгоритма нумеруются от 0 до n*, где n - это число блоков. *№1* - это блок, *с которого начинается* программа, остальные нумеруются произвольно, кроме *последнего блока. Он нумеруется 0*.

Следствие из теоремы о структурировании: любая программа может быть построена без использования операторов безусловной передачи управления (структурное программирование назвали *программированием без go to*).

К конструкция структурного

программирования можно

2) Вводится *дополнительная целочисленное переменная*, которое имеет смысл программного счётчика.

3) Каждый функциональный блок, *не связанный* с передачей управления, заменяется его *эквивалентом*, состоящим из *этого*

отнести ещё конструкций,

*расширить*

несколько то есть

*базисное*

*блока и следования за ним присваивания счётчику номера следующего функционального блока*.

4) Каждый функциональный блок, выполняющий действия, *связанные* с передачей управления, заменяется *эквивалентом*, который *не занимается реальной передачей управления*, а

*множество.*

1) Множественный выбор Множественный выбор - это

обобщение конструкции двоичного

*присваивает* *счетчику* *номера* *функциональных* *блоков, которым должно передаваться управление.*

5) Строится программа

int L = 1;

решения на случай множественного исхода теста. Конструкция множественного выбора сводится к

конструкции с использованием только

while (L) конструкции двоичного выбора и

switch (L) {

case 1: <выполнение блока #1>; break;

case 2: <выполнение блока #2>; break;

...

case n: <выполнение блока #n>; break;

}

<выполнение блока завершение программы (#0)>;

}

Схема в конспекте

Конструкции структурного программирования ⇒идея укрепления

функциональных блоков ⇒ дальнейшее *построение программы из*

следования. (Схема в конспекте)

2) Цикл с постусловием

Цикл с постусловием отличается от цикла с предусловием только тем, что решение о необходимости очередного повторения цикла принимается после выполнения функционального блока.

(Схема в конспекте)

*функционально законченных блоков*

Хорошо структурированная программа - это программа,

построенная из конструкций.

Эти 2 конструкции *повышают удобства и не нарушают структурности* *программы*. Основание использования этих конструкций то, что они сводятся к изначально базисному множеству.

20. Структурное программирование. Основные принципы и *Многократное* выполнение определенного функционального блока конструкции. с принятием решения о *необходимости очередного повторения* в

Структурное кодирование - это кодирование с использованием *только базисных конструкций*.

Если в языке программирования нет базисных конструкций, то их можно *моделировать (имитировать)*.

Положительные особенности структурного программирования:

зависимости от результатов некоторого теста проводимого *каждый раз перед выполнением этого блока*.

Используется неявные безусловный переход. 4) Множественный выбор

Множественный выбор - это обобщение конструкции двоичного

1) Использование конструкций структурного решения на случай *множественного исхода теста*. Конструкция

программирования при алгоритмизации *повышает ясность* алгоритма и делает его более стройным логически.

множественного выбора сводится к конструкции с использованием только конструкции двоичного выбора и следования.

2) Использование структурного кодирования позволяет строить 5) Цикл с постусловием

программы, для которых Цикл с постусловием отличается от цикла с предусловием только a) *Логика их функционирования может полностью* тем, что решение о *необходимости очередного повторения* цикла

*соответствовать логике алгоритмов*, построенных на основе конструкции структурного программирования

b) Существенно *упрощается анализ программ*, так как

принимается *после выполнения функционального блока*.

Схемы в конспекте + подробные преобразования и обоснования ОСНОВНЫЕ ПРИНЦИПЫ - соответствие структурной теореме:

*выполнение действий* в программе, может строго использование только базисных конструкций (без go to), объединение в *соответствовать порядку следования конструкций* функциональные блоки с единственным входом и единственным структурного программирования в *тексте алгоритма* выходом ??

(программы).

3) Из 1 и 2 пункта можно сделать выводы:

a) существенно уменьшается среднее число ошибок при кодировании

b) улучшается проработка структуры программы и в следствии, получается более оптимальный код

c) существенно возрастает скорость отладки программ и производительность труда программиста

Мысль 1:

Сначала писать код как хочется, а потом *преобразовывать код к структурному*.

Мысль 2:

Возможно использование других конструкций, применение которых приведёт к более качественному ПО. (но люди еще не открыли таких конструкций)

Конструкции структурного программирования: 1) Следование

Это происходит *прозрачно от программиста*, то есть блоки выполняются *в порядке появления в программе*.

2) двоичное “бинарное” решение

*Выбор одного из двух функциональных блоков* в зависимости от *результатов некоторого теста*.

3) Цикл с предусловием

21. Модульное программирование как составная часть 2. Можно переписать реализацию модуля и от этого ничего не структурного подхода к разработке программного обеспечения. поменяется для прикладного программиста. То есть в Понятие инкапсуляции. использваниии модуль останется таким же (если будет то е *Возможность объединения конструкций* структурного самый интерфейс на входе и на выходе). Отделяется интерфейс

программирования в более крупные блоки ведёт к мысли использования от реализации

функционально законченных, относительно независимых 3. Облегчение труда для программиста, использующего данные

функциональных блоков - модулей или программных единиц.

Обычно такие модули реализуют *функционально законченные*, *самостоятельные и почти независимые единицы* и могут располагаться независимо от других блоков, а также неоднократно вызываться в различных местах программы, в том числе из других программных единиц.

Объединение в функциональные блоки *экономит память, но ухудшает быстродействие.*

Модуль - подпрограмма:

*● процедура (ничего не возвращает) ● функция (возвращает значение)*

Модульное программирование = процедурное программирование Иногда модуль - это *хранилище процедур*, новых типов и т. д.

(типа библиотеки - в Pascal - модуль, в C/C++ - файл, в Java - пакет) Модульное программирование - *инструмент*, который на уровне

кодирования *реализует* *идеи* *структурного* *подхода* к программированию

*Детальный механизм* действия функциональных модулей для программирования с использованием этих модулей *не важен*. *Детали скрытые от разработчика* незанятого разработкой данного модуля, и это хорошо. Программист использует интерфейс, то есть *ему известен только заголовок.*

Функциональный блок или модуль реализует представление абстракции действий, *скрывая* ненужные извне *подробности* своей реализации.

*Сокрытие информации*, *изолирующее* скрываемые *детали* - это инкапсуляция (то есть изоляция от внешней среды).

Инкапсуляция в модульном программировании:

1) Существенно *повышает надёжность* и *эффективность* программ

2) Обеспечивает *лёгкость создания библиотек* подпрограмм -объединение модулей *по функциональному признаку* (по назначению), а значит *лёгкость построения программ из готовых функциональных блоков*.

Плюсы инкапсуляции (с лекции, относится к модульному программированию):

1. Нет необходимости вникать в детали реализации модуля

модули

4. Повышение безопасности. Данные и действия, спрятанные в модули не ломаются извне.

Достоинства модульного программирования:

● Повышение *эффективности разработки*. Большую программу могут писать одновременно несколько программистов;

● можно создавать *библиотеки* наиболее употребительных модулей;

● *Легкость тестирования*. Появляется много естественных контрольных точек для отладки проекта;

● Модульную программу *легче* *сопровождать* *и* *модифицировать*. Функциональные компоненты могут быть изменены, переписаны или заменены без изменений в остальных частях.

Недостатки модульного программирования:

● возрастает размер требуемой оперативной памяти;

● увеличивается время компиляции, загрузки и выполнения программы; ● довольно сложными становятся *межмодульные* интерфейсы.

с 4 курса:

Основные преимущества модульного программирования заключаются в следующем:

1. Модульность и возможность повторного использования: Каждый модуль имеет четко определенную цель и функциональность. Модули можно повторно использовать в различных программах или проектах, экономя время и усилия на разработку. Такая возможность повторного использования также улучшает сопровождаемость, поскольку изменения, внесенные в модуль, затрагивают только этот модуль, минимизируя влияние на остальную часть программы.

2. Абстракция и инкапсуляция: Модули обеспечивают уровень абстракции, скрывая внутренние детали и раскрывая только четко определенный интерфейс. Это позволяет другим частям программы взаимодействовать с модулем без необходимости знать его внутреннюю реализацию.

Инкапсуляция гарантирует, что внутренняя работа модуля скрыта и может быть доступна только через определенный интерфейс, что повышает безопасность кода и уменьшает количество зависимостей.

3. Удобство сопровождения и отладки: Модульный код легче поддерживать и отлаживать, поскольку изменения или исправления могут быть направлены на конкретные модули, не затрагивая всю программу. Это упрощает процесс отладки, изолируя проблемы от конкретных модулей, что облегчает их выявление и устранение.

22. Нисходящее проектирование как составная часть 2) *Вероятность появления различных ошибок* в программе структурного подхода к разработке программного обеспечения. существенно *понижается*.

Нисходящее тестирование. 3) *Организация* *и* *планирование* *работы* программистов

Если *функциональные блоки*, из которых надо строить хорошо структурированную программу, *могут объединяться* в эту программу с

*упрощается* (крупные проекты, разрабатываются в команде) 4) *Структурированность программ улучшается*

помощью укрупнение *с целью повышения степени инкапсуляции* 5) Дополнительно *уменьшается вероятность концептуальных действий*, реализуемых в данных модулях, то при соблюдении *и логических дефектов*, потому что яснее просматривается некоторых условий возможен *обратный процесс* - разукрупнение, то общая структура проекта и “узкие” места реализации есть возможно *развертывание единственного функционального блока* - (“бутылочное горлышко”). Обычно такие “узкие” места - это модуля реализующего всю программу *в* *набор* *комбинируемых* взаимодействие между модулями.

*функциональных блоков* различной степени детализации реализующих определенный функционал.

Проектирование:

Восходящее (сначала простые ФБ, из которых получаются более

Концептуальные дефекты - это такие дефекты, из-за которых *ломается идея проекта* и не удовлетворяется запрос заказчика

Логические дефекты менее значимые, они не рушат концепции проекта, но нарушают логику работы

сложные) 6) Значительно *раньше* можно *начать* *кодирование*

Нисходящее (Сначала крупные ФБ, решающие крупные задачи, которые разбиваются на более простые ФБ) - *сначала представить всю программу как один ФБ - черный ящик, а потом постепенно разбивать*

*программного комплекса в целом* и даже его *тестирование*, чем при нисходящем проектировании.

Нисходящее тестирование.

*на модули* На место не готовых (нереализованных или неадекватно

Схемы в конспекте

*Процесс разбиения крупных модулей на более мелкие* с различной степенью детализации называется

*Результат* *пошаговой* *д* декомпозиция.

работающих) модулей включаются *заглушки - stubs* (в конспекте не записала, но исходя из опыта, заглушки возвращают заранее заданные значения либо всегда одинаковые, либо в зависимости от значений входных параметров. У него на лекции: либо заглушки ничего не выполняют, либо выводят информационные сообщения. Сложнее если

1) На момент выполнения начальных этапов пошаговой модули зависят друг от друга , тогда и вид заглушки становится

детализации *подробности реализации каждого модуля могут быть пока ещё неизвестны*, это не мешает обдумывать общую структуру проекта.

сложнее)

Минусы нисходящего проектирования:

1) *тяжело выполнять тестирование нижних модулей*, так как

2) Наибольшее внимание уделяется взаимодействию между они зависят от верхних модулей.

модулями и их взаимосвязью, а не особенностям их Если не готовы средние модули а верхние и нижние готовы, то

реализации. Некоторые модули *уже могут существовать в библиотечной реализации*

заглушки для средних модулей более сложные, так как они должны имитировать передачу управления от верхних модулей нижним.

3) *Логика нисходящего проектирования соответствует логике* Драйверы (drivers) - это модули, которые обеспечивает

*человеческого мышления* при обдумывании сложных задач от крупного и общего к мелкому и частному - принцип

необходимые условия для тестирования нижних модулей (когда речь идет об интеграционном тестировании)

дедукции. 2) *Заглушки и драйверы требуют усилия для своей* Схема межмодульных связей (СМС) - иногда связи по данным (*что реализации*, но не являются реальными модулями,

*куда* *передается*), иногда связи по управлению (*что* *откуда* следовательно, *не выносят вклад в итоговый результат*.

*вызывается*)

Словесное описание алгоритма - это псевдокод. Преимущества нисходящего проектирования.

1) Соответствует *человеческой логике*

Иногда смешивают восходящее и нисходящее проектирование, если нижние модули достаточно просты в реализации (раньше называли метод “сендвича”).

23. Понятие объектно-ориентированного подхода к Конспект:

разработке программного обеспечения, предпосылки Представление предметной области:

возникновения и преимущества применения. (короче тут пиздец, Сущность (entity) - объекты, явления понятия и процессы и т.д., я не разобралась и смешала все в одну кучу) + мало про плюсы которые существуют в предметных областях. Степень детализации

Для лучшего понимания: Павловская:

Объектно-ориентированное программирование (ООП) — это технология, возникшая как реакция на очередную фазу кризиса программного обеспечения, когда методы структурного программирования уже не позволяли справляться с растущей сложностью промышленного программного продукта. Следствия — срыв сроков проектов, перерасход бюджета, урезанная функциональность и множество ошибок.

Существенная черта промышленной программы — ее сложность: один разработчик не в состоянии охватить все аспекты системы, поэтому в ее создании участвует целый коллектив. Следовательно, к первичной сложности самой задачи, вытекающей из предметной области, добавляется управление процессом разработки с учетом необходимости координации действий в команде разработчиков.

Так как сложные системы разрабатываются в расчете на длительную эксплуатацию, то появляются еще две проблемы: сопровождение системы (устранение обнаруженных ошибок) и ее модификация, поскольку у заказчика постоянно появляются новые требования и пожелания. Иногда затраты на сопровождение и модификацию сопоставимы с затратами на собственно разработку системы.

Способ управления сложными системами был известен еще в древности — divide et impera (разделяй и властвуй). То есть выход — в декомпозиции системы на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. Но если в рамках структурного подхода декомпозиция понимается как разбиение алгоритма, когда каждый из модулей системы выполняет один из этапов общего процесса, то ООП предлагает совершенно другой подход.

Суть его в том, что в качестве критерия декомпозиции принимается принадлежность ее элементов к различным абстракциям проблемной области. Откуда же берутся эти абстракции? Исключительно из головы программиста, который, анализируя предметную область, вычленяет из нее отдельные объекты. Для каждого из этих объектов определяются свойства, существенные для решения задачи. Затем каждому реальному объекту предметной области ставится в соответствие программный объект.

иванова ничушкина пугачев:

В теории программирования ООП определяется как технология создания сложного программного обеспечения, которая основана на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа {класса), а классы образуют иерархию с наследованием свойств.

Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений (рнс 1.6).

Основное достоинство ООП - сокращение количества межмодульных вызовов и уменьшение объемов информации, передаваемой между модулями, по сравнению с модульным программированием. Это достигается за счет более полной локализации данных и интегрирования их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы.

Основной недостаток ООП - некоторое снижение быстродействия за счет более сложной организации программной системы.

сущности зависит от задачи.

Уровень абстракции - это уровень детализации сущности. Абстракция - это упрощенное представление реальной сущности с

характеристиками, важными для конкретного контекста.

Чем богаче средство языка программирования, позволяющее реализовать абстракцию тем выше уровень языка программирования.

Подходы к представлению предметной области:

Любой язык программирования можно рассматривать с 2 точек зрения:

1 - концепция данных - как данные устроены, какие действия можно выполнять над этими данными

2 - концепция действий - во главе действия, а потом уже какие данные они обрабатывают

Богаче концепция данных.

Структурный подход относится к концепции действий.

Сущность имеет что-то связанное с данными и что-то связанное с действиями.

Абстракция данных, абстракция действий.

ООП и структурный подход зародились примерно в одно и тоже время, но ООП развивалось медленнее по следующим причинам:

Концепция действий (и абстракция действий) проще, поэтому развивалась быстрее. Идеи концепции действий можно реализовать на языках любого уровня, в том числе низкого. Развитие концепции действий быстро дало положительный результат.

Развитие концепции данных шло медленнее, но наработок стало больше за счёт разнообразия форм данных.

Инкапсуляция скрывает в первую очередь действия.

Развитие абстракций:

Модули - это 1 шаг абстракции.

Инкапсуляция на уровне модуля требует описание данных, используемых в модуле, особенно для взаимодействующих модулей.

Минусы:

1) Зависимость модуля от внешней среды

2) Снижение безопасности, так как разные данные используются разными модулями

У модулей нет определенности по смысловому наполнению данных, с которыми работает модуль. Это следствие того, что при структурном подходе во главе угла, концепция действий, а данные абстрактное нечто.

Чем сложнее данные и чем крупнее проект, тем сложнее работать с

Центральные понятия ООП:

В предметной области решаемой задачи есть некоторые сущности. Эти сущности воплощаются в программе в виде объекта. Объект - это объединение данных с алгоритмами их обработки.

концепцией действий. Программа - объединение объектов. Объект находится в Абстрактный тип данных (АТД) - это пользовательский тип данных некотором состоянии (как в реальной жизни). Как и в жизни, объекты вместе с набором действий над ними. могут обладать некоторыми свойствами, причем свойства и состояния Абстрактный тип данных - это 2 шаг абстракции. близко связаны между собой, можно считать что это одно и то же. То Экземпляр - это конкретные перемены какого-либо типа. есть каждое свойство объекта принимает определенное значение, Библиотеки модулей и абстрактных типов данных - это 3 шаг формируя состояние объекта. Изменение состояния связано с

абстракции.

Краткая историческая справка по ООП:

изменением значений свойств. Иногда эти понятия объединяют, а иногда разделяют.

Иногда говорят, что объекты могут взаимодействовать между

Основоположники идей ООП - это Вирт Н., Хоар Ч. Э. Р и Дал О. собой через механизм сообщений (модель обмена сообщениями И. ООП базируется на структурном подходе является его развитием. реализована не везде)

Дал разработал Симула-67 (1967 год) - первый Объекты обладают некоторой типизацией, то есть объединяются в объектно-ориентированный язык программирования (но на некоторые группы объектов, которые называются термином класс. Тип примитивном уровне).В силу слабой разработанности концепции данных объекта - это класс (тип данных), а конкретный объект - экземпляр этого язык не получил широкого применения. класса. С каждым классом связан определенный набор действий -

Концепция данных долго развивалась (как и идеи ООП).

В середине 80-х Страуструп на основе языка программирования С разработал язык C with classes (C с классами) - C++.

Инкапсуляция в ООП:

методов, которые доступны объектам этого типа (класса). Объекты могут выполнять некоторые действия, которые определяют его поведение.

Обычно говорят, что объект обладает некоторыми свойствами и поведением.

На уровне воплощения в ЯП:

1) Вывод инкапсуляции на новый уровень - тесное объединение ● Свойства - поля

данных с методами их обработки, чтобы ● Поведение - методы (действия присущие данному классу и

a) наполнить смыслом связи между данными и действиями над ними, которые ранее были формальными

b) повышение уровня абстракции

2) Достичь в большей степени инкапсуляции для независимости от внешней среды, то есть повышение безопасности.

Чего хотели добиться в ООП:

его экземплярам)

Основная идея ООП (отличительная особенность, новая трактовка):

В структурном подходе был блок действий, на вход которого подавались данные, над которым выполнялись данные действия, то есть действия, выполнялись над объектами данных.

Совершенствование абстракции, совершенствование отображения В объектно-ориентированном подходе сам объект выполняет

тех предметных областей, которые фигурируют в задачах, чтобы модели более полно отражали реальность и были более адекватными.

действия, которые присущи его поведению. Данная трактовка более содержательна, а следовательно повышает уровень абстракции. Это

Вывод инкапсуляции на новый уровень повышает уровень адекватности представления сущности и, Вывод абстракции на новый уровень соответственно, уровень абстракции модели

ООП - модель или парадигма, которая говорит, что программа представляет собой совокупность объектов. Все есть объект, а объект отражает сущность.

Преимущества:

1. Модульность и повторное использование кода:

Один раз написанный и отлаженный класс можно использовать в разных частях проекта, а также в других проектах.

2. Гибкость и расширяемость:

Можно изменить внутреннюю реализацию объекта, не затрагивая другие части кода, если интерфейс остается неизменным.

Добавление нового функционала часто сводится к созданию нового подкласса с нужными методами.

3. Повышение надежности и безопасности:

Инкапсуляция защищает данные: Прямой доступ к внутренним данным объекта ограничен, что снижает риск случайных ошибок.

4. Упрощение командной работы:

Разные разработчики могут работать над разными модулями системы, не мешая друг другу.

РЕЗЮМЕ:

Преимущества ООП проявляются в крупных проектах. Изначально ООП разрабатывалась для того, чтобы облегчить труд программистов.

На этот подход возлагали много надежд, но потом популярность спала, так как стало появляться много корпоративных языков.

Предполагалось, что ООП уменьшит сложность и объем проектов, но по факту сильного облегчения не произошло. ООП помогает именно в крупных проектах. В небольших проектах ООП не дает никаких преимуществ, скорее наоборот неоправданно увеличивает объем и сложность кода. Происходит неэффективное использование тяжеловесных средств. (“стрелять из пушки по воробьям”)

Использовать принципы и подходы ООП имеет смысл, когда сущности выстраивают некоторую четко выраженную иерархию (случаи. когда можно выстроить схему наследования свойств). Если

24. Основные идеи и принципы объектно-ориентированного подхода к разработке программного обеспечения. - маловато?

1) Абстракция

Абстракция - это *упрощенное представление реальной сущности* с характеристиками важными для конкретного контекста.

Шаги абстракции: 1. Модули

2. Абстрактный тип данных 3. Библиотеки модулей и АТД

В ООП абстракции представляются в виде классов и их экземпляров - объектов. В объектно-ориентированном подходе сам объект выполняет действия, которые присущи его поведению. Данная трактовка более содержательна, а следовательно повышает уровень абстракции.

2) Инкапсуляция

*Сокрытие информации*, изолирующее скрываемые детали - это инкапсуляция (то есть изоляция от внешней среды).

В отличие от структурного подхода в ООП происходит инкапсуляция не только действий, но и данных, что повышает уровень безопасности.

Инкапсуляция – это принцип, согласно которому любой класс должны рассматриваться как «черный ящик»: пользователь класса видит только интерфейс (т.е. список декларируемых свойств и методов) и не вникает во внутреннюю реализацию.

Польза: Защита данных от случайных изменений, упрощение взаимодействия между объектами.

Пример: Управление телевизором с помощью пульта – вы не знаете, как он устроен внутри, но можете переключать каналы.

иерархия классов совсем не просматривается, то достаточно ограничится структурным подходом.

3) Наследование

Наследование - *получение* каким-либо компонентом программы

*свойств* *некоторого* *другого* *компонента* в соответствии со *специальными* *отношениями*, существующими между этими компонентами.

Наследование объектов позволяет выстроить *иерархию объектов*. *Наверху* иерархии располагаются объекты, которые и инкапсулируют о*бщие свойства* (без особой детализации), а *внизу иерархии - более детализированные объекты.*

Создание новых объектов (подклассов) на основе существующих (суперклассов), наследуя их атрибуты и методы.

Польза: Повторное использование кода, создание иерархий объектов.

Пример: Класс "Собака" наследует атрибуты класса "Животное" (имя, возраст) и добавляет свои ("порода").

4) Полиморфизм 25. Инкапсуляция как один из принципов

Полиморфизм - определение *единых по своей идентификации* (то есть вызов происходит одинаково) (часто одинаковых и по смыслу) действий *реализующихся по-разному*.

Возможность объектов разных классов реагировать на одинаковые запросы по-разному.

Польза: Гибкость кода, возможность обрабатывать объекты разных типов единообразно.

объектно-ориентированного подхода к разработке программного обеспечения. - точно ли то ?

Инкапсуляция в общем смысле:

*Сокрытие информации*, изолирующее скрываемые детали - это инкапсуляция (то есть изоляция от внешней среды).

Инкапсуляция в структурном подходе:

Пример: Метод "говорить()" у объектов "Кошка" и "Собака" будет издавать разные звуки.

Структурный подход относится к концепции действий. Инкапсуляция в структурном подходе скрывает в первую очередь действия.

Развитие инкапсуляции при повышении степени абстракции:

I. Инкапсуляция на уровне модуля

*Инкапсуляция на уровне модуля требует описание данных*, используемых в модуле, особенно для взаимодействующих модулей.

*Минусы:*

1) Зависимость модуля от внешней среды и друг от друга

2) Снижение безопасности, так как разные данные используются разными модулями

При инкапсуляции действий, то есть инкапсуляции на уровне модулей, построение библиотек происходит по функциональному принципу. То есть объединяются модули, направленные на выполнение определенных типов действий, например, библиотека математических операций.

У модулей нет определенности по смысловому наполнению данных, с которыми работает модуль. Это следствие того, что при структурном подходе во главе угла, концепция действий, а данные абстрактное нечто. Смысл передаваемых данных никак не контролируется (кастрюльке без разницы, что в нее кидать), за этим должен следить разработчик. В крупных проектах это может вызвать затруднение.

II. Абстрактный тип данных

Чем *сложнее данные* и чем крупнее проект, тем *сложнее работать с концепцией действий*. (сложно контролировать смысл данных, передаваемые в модули )

У каждого типа данных есть свой набор действий, для обработки этих данных.

Абстрактный тип данных (АТД) - это пользовательский тип данных (обычно достаточно сложно устроенный) вместе с набором действий над ними.

Из Ахо: АТД - это математическая модель представления данных вместе совокупностью операторов или операций, определенных в рамках этой модели.

Из Орлова: АТД - это синтаксический контейнер, а контейнер понимается как структура данных, некое описание, включающее определение данные некоторого типа и набор действий над ними

Тоже инкапсуляция, но она скрывает не только действия, но и ● данные ● `

|  |  |
| --- | --- |
| `private  protect | ` ( |
| ed |

приватный): Доступны только внутри класса.

` (защищенный): Доступны внутри класса, в

Если имеется библиотека для работы с АТД, например с деревом, подклассах и в пределах того же пакета (зависит от языка). то там есть представление самого АТД (дерева) на уровне данных и Геттеры и сеттеры (Getters & Setters): Специальные методы для набор операций над АТД, но детали при этом скрыты от пользователя получения (get) и изменения (set) значений приватных атрибутов.

библиотеки (инкапсуляция).

Абстрактный тип данных - это 2 шаг абстракции.

Инкапсуляция дает упрощение использования. Происходит отделение интерфейса от реализации.

III. Библиотеки модулей и абстрактных типов данных - это 3 шаг абстракции. (немного притянутый шаг)

Если есть сложные типы данных, требующие специфичные операции на ними, тогда можно объединить АТД и модули для работы с ними.

Инкапсуляция в ООП:

ООП базируется на структурном подходе является его развитием, в том числе и в принципе инкапсуляции. Вывод инкапсуляции на новый уровень. Более тесное объединение данных и действий над ними чем в АТД. Например, бинарное дерево и методы его обработки - это АТД, так как дерево может содержать разные по смыслу данные. В ООП есть конкретный смысл данных и конкретный смысл связей между ними.

Инкапсуляция в ООП:

1) Вывод инкапсуляции на новый уровень - тесное объединение данных с методами их обработки, чтобы

a) *наполнить смыслом связи* между данными и действиями над ними, которые ранее были формальными

*b) повышение уровня абстракции*

2) Достичь в *большей степени инкапсуляции* для независимости от внешней среды, то есть повышение *безопасности*.

Объект (сущность в данном контексте) - это объединение данных с алгоритмами их обработки.

Сущность имеет что-то связанное с данными и что-то связанное с действиями. Следовательно, правильная абстракция сущности должна одержать и *абстракцию действий, и абстракцию данных*

Абстракция в ООП реализуется в виде объекта, у которого есть состояние и свойства.

Обеспечивают контролируемый доступ к данным. Суть инкапсуляции в ООП:

Скрытие информации: Внутреннее устройство объекта скрыто от внешнего мира. Как он хранит данные или реализует свои методы – не важно для пользователя объекта.

Инкапсуляция позволяет изолировать модули от внешней среды.

Плюсы инкапсуляции (с лекции, относится к модульному программированию): 5. Нет необходимости вникать в детали реализации модуля

6. Можно переписать реализацию модуля и от этого ничего не поменяется для прикладного программиста. То есть в использваниии модуль останется таким же (если будет то е самый интерфейс на входе и на выходе). Отделяется интерфейс от реализации

7. Облегчение труда для программиста, использующего данные модули

8. Повышение безопасности. Данные и действия, спрятанные в модули не ломаются извне.

При правильном подходе к инкапсуляции использование классов предполагается через интерфейс. (Интерфейс - набор программно-аппаратных средств для обеспечения взаимодействия между чем-либо.)

Межклассовый интерфейс - средство сопряжения классов, доступные извне (аналог прототипа функций). По отношению к классам - это методы, доступные снаружи.

Считается, что все поля должны быть инкапсулированы (спрятаны от внешней среды). Геттеры и Сеттеры позволяют добраться до закрытых полей класса. Но геттеры и сеттеры нужны не для всех полей (например, они не нужны для чисто служебных полей).

Методы - средство организации инерфейса класса, а детали реализации скрыты за этим интерфейсом. Это хорошо, так как нет необходимости вникать в эти детали, а значит, мы можем переписать реализацию класса не меняя интерфейс класса. Это позволяет независимо проводить рефакторинг класса.

Как реализуется инкапсуляция:

Модификаторы доступа: В большинстве языков ООП используются ключевые слова, определяющие уровень доступа к членам класса (атрибутам и методам):

● `public` (публичный): Доступны *из любой части программы*.

26. Наследование как один из принципов 27. Полиморфизм как один из принципов

объектно-ориентированного подхода к разработке программного обеспечения. Варианты реализации наследования. маловато?

объектно-ориентированного подхода к разработке программного обеспечения. Варианты реализации полиморфизма. маловато?

Полиморфизм - определение *единых по своей идентификации* (то

Наследование - *получение* каким-либо компонентом программы есть вызов происходит одинаково), часто одинаковых и по смыслу *свойств* *некоторого* *другого* *компонента* в соответствии со действий *реализующихся по-разному*.

специальными отношениями, существующими между этими Удобно, чтобы похожие действия, которые только немного

компонентами.

Наследование - это возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка, добавляя при необходимости новые свойства и методы.

Главная цель: Избежать дублирование кода и выстроить иерархию с различной степенью детализации

отличаются назывались одинаково по отношению к разным сущностям. Например есть класс точка. Ее можно выводить на экран. Из

точек формируется линия, значит можно создать класс линия, которую также можно выводить на экран. По своей сути операция вывода та же, то есть мы ожидаем похожего поведения. Аналогично с классами, треугольник, многоугольник и т.д. Они реализуются по-разному, но

имеют одинаковый смысл, поэтому удобно чтобы их вызов был Такой подход существует не только в ООП. Например, свойства идентичный.

структуры определяются свойствами ее полей или например,

пользовательский тип, объявленный через typedef, получает свойства базового типа

Наследование объектов позволяет выстроить иерархию объектов, что особенно хорошо для крупных проектов.

Полиморфизм существовал до задолго до ООП (но в ООП стал богаче):

Пример:

2+3 и str1+str2

*Наверху* иерархии располагаются объекты, которые и *инкапсулируют общие свойства* (без особой детализации), а *внизу*

*иерархии - более детализированные объекты*. Это также способствует повышению уровню абстракции.

Родитель (прародитель) → потомки

*В потомках реализуется в поведении нереализованное в родителе.*

Варианты реализации полиморфизма:

*же метод в разных классах* (overriding)

Переопределение метода означает наличие двух методов с Наследование бывает (варианты реализации наследования): *одинаковыми аргументами, но разными реализациями*. Один из них

● простое - один *единственный* родитель будет существовать в родительском классе, а другой - в производном или ● множественное - *много* родителей (> 1). (множественное дочернем классе. Наследник может выполнять команду базового класса

наследование обычно порождает проблему «ромбического по своему

наследования»: при наследовании от двух потомков одного класса непонятно как разруливать многократное наследование одних и тех же его членов — полей, свойств и методов)

Схема в конспекте.

Бывает ромбовидное наследование. Схема в конспекте. Предок, родитель, родительский класс, надкласс, суперкласс.

↓

Потомок, производный класс, подкласс, субкласс.

Пример - вывод разных фигур

*реализацией в одном объекте* (overloading)

Перегрузка метода - это определение нескольких методов в *одном классе*, которые принимают *разное количество и типы параметров*. В этом случае фактический вызываемый метод определяется во время компиляции на основе количества и типов аргументов.

Выбирается правильный экземпляр функции в соответствии с

Есть языки, в которых все объекты - потомки некоторого вызовом.

стандартного родительского класса. Например, в Delphi все объекты Пример - вывод одной и той же фигуры по-разному наследуются от TObject. В C++ такого нет.