Содержание

Введение

1. История развития языков программирования
2. Основы объектно-ориентированного программирования
3. Классы
4. Конструкторы и деструкторы
5. Наследование
6. Указатель this
7. Друзья
8. Перегрузка операций
9. Константные объекты и константные методы

Литература

Введение

Важной вехой в развитии программирования явилось создание и широкое распространение языка С++. Этот язык, сохранив средства ставшего общепризнанным стандартом для написания системных и прикладных программ языка С (процедурно-ориентированный язык), ввел в практику программирования возможности нового технологического подхода к разработке программного обеспечения, получившего название “объектно-ориентированное программирование”. Внедрение в практику программирования объектно-ориентированной парадигмы дает развитие новых областей информатики, значительное повышение уровня технологичности создаваемых программных средств, сокращение затрат на разработку и сопровождение программ, их повторное использование, вовлечение в процесс расширения интеллектуальных возможностей ЭВМ. Объектный подход информационного моделирования предметных областей все более успешно применяется в качестве основы для структуризации их информационных отражений и, в частности, баз знаний.

1. История развития языков программирования

Первые компьютеры появились в конце Великой Отечественной Войны сначала в Соединенных Штатах Америки, а позже в СССР. Эти машины могли решать ограниченный класс задач.

Кодирование происходило на физическом уровне. Сначала программы хранились на перфокартах. Таким образом, процесс отладки программы занимал очень много времени. Позже появились магнитные ленты и магнитные диски, что облегчило задачу переносимости программ.

В конце 1950 – начале 1960 годов появились языки программирования Fortran(FORmula TRANslation) и Cobol(COmmon Business Oriented Language)-язык, ориентированный на выполнение общих экономических расчетов.

В 1960-70 годах было написано множество языков программирования. Почти каждый программист придумывал свой язык, мечтая увековечить свое имя.

В конце 1970-х появились Паскаль, Модула, Си, которые широко применялись.

В начале 1980-х широкое распространение получили персональные компьютеры. Примерно в это же время появился язык C++.

Естественно, что C++ более всего близок к языку C. Язык С полностью включен в C++, оставлены все возможности С как языка низкого уровня для выполнения наиболее сложных и универсальных программ. Другим источником вдохновения был язык Simula67; оттуда заимствованы концепции классов и производных классов с виртуальными функциями.

Название языка C++ возникло летом 1983 года. Более ранние версии, известные под именем “C с Классами”, используются с 1980 года. Первоначально язык возник в процессе создания программы событийно-управляемой симуляции, для которой идеально подошел бы язык Simula67, если бы не соображения эффективности. “C с Классами” использовался для основных проектов по симуляции только в программах, критичных по времени выполнения и объему используемой памяти. C++ впервые возник вне группы автора в июле 1983года, однако он уже тогда практически не отличался от современной версии языка.

Название “C++ “ было предложено Риком Масцитти и символизирует эволюционные изменения, произошедшие с языком C(“++” – обозначение оператора инкрементации в языке C).

Таким образом, эволюцию языков можно показать на схеме:

Коды Ассемблеры Языки высокого Объектно-ориентированное и уровня модульное программирование

2. Основы объектно-ориентированного программирования:

Объектно-ориентированное программирование – это самый высокоуровневый вид программирования в настоящее время. Здесь поддерживаются достижения предыдущих поколений и добавлены новые свойства. Эти новые свойства реализуют парадигму объектно-ориентированного программирования.

Объектно-ориентированное программирование на C++ основывается на следующих основных этапах разработки программ.

Первый этап заключается в выделении абстракций. Выделение абстракций означает анализ предметной области, для которой составляется программа, с целью определения основных объектов предметной области, их свойств, отношений между объектами, а также возможных операций над объектами и их составляющими.

Второй этап состоит в типизации объектов и синтезе абстрактных типов данных.

Этап предполагает определение новых производных типов данных и наборов специфических функций или операций, применяемых к этим типам данных таким образом, чтобы исключить возможность смешивания или взаимозамены различных типов.

Третий этап заключается в объектной декомпозиции как выделении подтипов или подобъектов и их составляющих для каждого из типов объектов.

Четвертый этап представляет собой композиционную иерархизацию объектов как выделение родовидовых и композиционных отношений над объектами.

В результате объектно-ориентированного подхода к проектированию программ процесс разработки программы превращается в процесс эволюционного программирования, который для внесения каких-либо изменений и дополнений в программу не требует кардинального пересмотра составляющих ее алгоритмов. Эволюционный способ программирования опирается на сохранение целостности объектов программы, то есть внесение изменений в программу не должно затрагивать внутреннюю организацию существующих в ней объектов.

Важным свойством объектно-ориентированных языков является возможность разработки на них программ, работающих в системах со сложными параллельными вычислительными процессами, изначально присущими техническим средствам вычислительной техники. Это свойство опирается на концепцию активных и неактивных объектов в период функционирования программы. Одновременная активность различных объектов становится возможной за счет их строгой типизации и закрытости для изменений другими объектами.

Язык программирования C++ обладает всеми основными свойствами языков объектно-ориентированного программирования и существенно отличается по своей концепции от базового языка C.

Существует несколько принципов, лежащих в основе языка C++:

1. Инкапсуляция – это объединение производного типа данных с набором функций, используемых при работе с этим типом, в единый класс. При этом функции, включенные в класс, называются методами класса, данные – элементами данных, а конкретные представители класса – объектами класса. Закрытие данных и методов оперирования этими данными происходит таким образом, чтобы обращаться можно было бы только определенным объектам (в данном случае только объектам этих типов).

2. Наследование – это способность одних классов заимствовать основные свойства других классов, в частности – методы классов и элементы данных. Класс, наследующий свойства, называют производным, а класс, предоставляющий свои свойства для наследования, - базовым. Механизм наследования позволяет создавать иерархию классов, то есть многоуровневую систему классов, связанных между собой отношением наследования.

3. Полиморфизм – это возможность определения функции, работающей с различными по типу данных списками параметров в пределах какого-либо одного вида алгоритмов. Такие функции называются обычно виртуальными и проектируются как некоторое семейство одноименных функций, работающих с различными типами данных. Механизм, реализующий выбор какой-либо конкретной функции из определенного семейства, носит название механизма позднего связывания, поскольку может быть использован в процессе выполнения готовой программы.

3. Классы

Центральным понятием объектно-ориентированного программирования является понятие класса. Именно он реализует основные свойства: инкапсуляцию, наследование, полиморфизм.

Класс представляет собой тип, определяемый пользователем. Этот тип включает в себя совокупность полей данных и функций для работы с этими полями.

Класс объявляется:

class <имя класса>[: public <имя базового класса>]

{поля данных

спецификатор доступа

функции};

Отнесение переменной к какому-то типу данных определяет память, выделяемую для этой переменной и набор операций и функций, применяемых к таким переменным. Реализация этих принципов для типов, определяемых пользователем, осуществляется при помощи классов.

Спецификаторы доступа

Существует ряд соображений, по которым было бы целесообразно ограничить доступ к элементам данных класса. К наиболее важным из них относятся следующие:

- ограничение доступа к данным класса рамками тех функций, которые включены программистом в этот класс, позволяет локализовать программные ошибки практически до начала работы программы;

- описание класса в этом случае позволяет пользователям классов более просто знакомиться с новыми библиотеками классов;

- при ограничении доступа упрощается корректировка программ, поскольку для их изменения достаточно скорректировать описание класса и функции, являющиеся его членами, не внося изменений в те места программы, где используются объекты класса;

- функциональное разграничение классов делает возможной разработку программ, использующих концепцию параллельных процессов.

public

Этим спецификатором помечается группа данных и функций, которые доступны другим функциям программы.

protected

Помечаются защищенные данные и, возможно, функции, если есть необходимость. Эти элементы доступны только функциям - членам данного класса и производных от него классов, то есть тех классов, которые объявят себя приемниками данного.

private

Служит для задания данных и функций, доступных только функциям данного класса. Это частные данные.

По умолчанию элементы считаются частными (private) для класса и открытыми (public) для структуры (объединения также относят к классам).

Классы лучше определять в файле с расширением .h, а реализацию в файле с тем же именем, но с расширением .cpp или .c. Чаще всего класс по одиночке не определяется, а создаются библиотеки.

Для иллюстрации рассмотрим пример класса, который задает координату на экране:

class Location

{ int x;

int y;

public:

Location (int \_x, int \_y); //конструктор

void setx(int nx);

void sety(int ny);

int Getx()

{return x;}

int Gety()

{return y;}

};

В данном примере, использовав спецификатор public, мы сделали открытыми для других функций методы, описанные в классе.

Определить функции – члены класса можно внутри описания класса или за его пределами. В первом случае функция считается встраиваемой. Встраиваемая функция характерна тем, что компилятор C++, обрабатывая вызовы этой функции в программе, заменяет их не на вызов функции как подпрограммы, а непосредственно на объектный код, соответствующий определению этой функции. Вследствие сказанного, программист должен принимать во внимание, что встраиваемые функции, как правило, имеют короткие определения. В качестве примера можно привести определение функций Getx() и Gety().

Для определения функции – члена класса за пределами описания класса необходимо определить ее где-либо в программе после определения класса, членом которого она является.

void Location :: setx(int nx)

{x=nx;}

void Location :: sety(int ny)

{y=ny;}

Location :: Location (int \_x, int \_y)

{x=\_x; y=\_y;}

Операция разрешения контекста (::) позволяет указать компилятору, к какому из классов принадлежит определяемая функция.

Имя класса в определении пишется для того, чтобы компилятор однозначно определил к какому классу принадлежит данная функция, так как функции – члены различных классов могут иметь одинаковые имена.

При определении классов не происходит реального выделения памяти под объекты этого класса, а создаются лишь новые производные типы данных, для которых будут использоваться функции – члены класса.

Для того, чтобы начать работу с реальными объектами какого-либо класса, эти объекты необходимо сначала определить. При этом в программе необходимо указать имя класса, объект которого должен быть создан, а также имя самого объекта. У каждого из классов может быть произвольное число объектов.

## 4. Конструкторы и деструкторы

Конструктором называется функция-член класса, которая выделяет память под поля данных класса и производит их инициализацию, т.е. задает начальные значения в месте объявления переменных.

Имя конструктора совпадает с именем класса. Например, в классе Location конструктор имеет следующий вид: Location (int \_x, int \_y).

Конструктор не возвращает никакого значения, даже void.

Одним из важных свойств конструктора является его автоматический вызов при описании любого объекта какого-либо класса, использующего конструктор, что снимает с программиста задачу своевременного отслеживания инициализации вновь вводимых объектов.

В общем случае конструкторы классов могут иметь списки параметров, которые могут потребоваться при инициализации. При этом программист будет обязан задать список инициализации при описании каждого нового объекта.

Конструкторов в классе может быть много. В этом случае реализуется механизм перегрузки функции.

Если конструкторы не объявлены, компилятор сам создает конструктор без параметров по умолчанию.

Объявление объектов можно проиллюстрировать следующим образом:

void main (void)

{Location NK(0,0), KK(10,10), \*PL;

cout<<KK.Getx(); //возвращаемое значение : 10

PL=&NK;

cout<<PL->Gety(); //возвращаемое значение : 0

}

Здесь при объявлении NK(0,0) и KK(10,10) неявно вызываются конструкторы.

cout<<KK.Getx() обращение идет через переменную.

cout<<PL->Gety() обращение идет через указатель.

Конструктор копий

{Location A(1,1),B,D=A;

… }

Сначала создается объект D и он инициализируется значением объекта A. Для инициализации нужно явно определить конструктор.

В конструкторе копий в качестве параметра используется простая или константная ссылка на объект.

Location::Location([const]Location &S)

{x=S.x; y=S.y}

Для каждого из объектов класса при очистке памяти компилятором создается деструктор по умолчанию. Определяется деструктор следующим образом: ~ имя. Имя деструктора совпадает с именем класса, но с символом ~ (тильда) в начале.

Деструктор решает обратную конструктором задачу, т.е. очищает память.

Если в конструкторе объекта запрашивается динамическая память или открывается файл, то при уничтожении объекта необходимо предусмотреть действия по очистке памяти и закрытию файла. В этом случае пользователю необходимо определять деструктор. Этот деструктор будет вызываться при выходе объекта из области видимости.

Локальные объекты удаляются тогда, когда они выходят из области видимости. Глобальные объекты удаляются при завершении программы.

5. Наследование

Применительно к C++ наследование – это механизм, посредством которого один класс может наследовать свойства другого. Наследование позволяет строить иерархию классов, переходя от более общих к более специальным.

Класс, свойства и поведение которого наследуются, называется базовым классом.

Класс, который наследует называет, называется производным классом.

Обычно процесс наследования начинается с задания базового класса. Базовый класс определяет все те качества, которые будут общими для всех производных от него классов. В сущности, базовый класс представляет собой наиболее общее описание ряда характерных черт. Производный класс наследует эти общие черты и добавляет свойства, характерные только для него.

Наследование, при котором указывается один базовый класс, называется простым.

Если указываются несколько классов, то наследование называется множественным.

Объявление выглядит следующим образом:

class имя класса : public имя базового класса

Например, class D: public A

{ … }

После имени класса D имеется двоеточие, за которым следует ключевое слово public и имя класса A. Для компилятора это указание на то, что класс D будет наследовать все компоненты класса A. Само ключевое слово public информирует компилятор о том, что, поскольку класс A будет наследоваться, значит, все открытые элементы базового класса будут также открытыми элементами производного класса. Однако все закрытые элементы базового класса останутся закрытыми и к ним не будет прямого доступа из производного класса. Причина, по которой закрытые члены класса становятся недоступными для производных классов – поддержка инкапсуляции. Если бы закрытые члены класса становились открытыми просто посредством наследования этого класса, инкапсуляция была бы совершенно несостоятельна.

При множественном наследовании объявление выглядит так:

class D: public A [, public C]

{ тело класса D}

Рассмотрим пример:

enum Bool

{false, true}; //константы сводятся к int. Они изменяются с шагом равным единице.

class Point: public Location

{protected:

Bool vis;

public:

Point (int \_x, int \_y);

void Show();

void Hide();

};

Point::Point (int\_x, int\_y) : Location(\_x, \_y)

{vis=false;}

Здесь класс Point наследует свойства базового класса Location.

Наследование и контроль доступа

Спецификатор доступа определяет то, как элементы базового класса наследуются производным классом. Если спецификатором доступа наследуемого базового класса является ключевое слово public, то все открытые члены базового класса остаются открытыми и в производном. Если спецификатором доступа наследуемого базового класса является ключевое слово private, то все открытые члены базового в производном классе становятся закрытыми. В обоих случаях все закрытые члены базового класса в производном классе остаются закрытыми и недоступными.

Важно понимать, что если спецификатором доступа является ключевое слово private, то хотя открытые члены базового класса становятся закрытыми в производном, они остаются доступными для функций – членов производного класса.

Доступ к полям базового класса в производном классе может быть сохранен или ужесточен, но никогда не может быть облегчен. Чтобы нагляднее представить себе этот принцип, обратимся к таблице:

|  |  |  |
| --- | --- | --- |
| Доступнаследования | Доступ компонентовв базовом классе | Доступность компонентовбазового класса впроизводном классе |
| public | privateprotectedpublic | Нет доступаprotectedpublic |
| protected | privateprotectedpublic | Нет доступаprotectedprotected |
| private | privateprotectedpublic | Нет доступаprivateprivate |

6. Указатель this

Когда функция, принадлежащая классу, вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого функция вызвана. Этот указатель имеет фиксированное имя this и незаметно для программиста (“тайно”) определен в каждой функции класса следующим образом:

имя\_класса \* const this=адрес обрабатываемого объекта;

Имя this является служебным (ключевым) словом. Явно описать или определить указатель this нельзя и не нужно. В соответствии с неявным определением this является константным указателем, т.е. изменить его нельзя, однако в каждой принадлежащей классу функции он указывает именно на тот объект, для которого функция вызывается. Говорят, что указатель this является дополнительным (скрытым) параметром каждой нестатической компонентной функции. Другими словами, при входе в тело принадлежащей классу функции указатель this инициализируется значением адреса того объекта, для которого вызвана функция. Объект, который адресуется указателем this, становится доступным внутри принадлежащей классу функции именно с помощью указателя this. При работе с компонентами класса внутри принадлежащей классу функции можно было бы везде использовать этот указатель.

Таким образом, каждая нестатическая функция – элемент класса имеет доступ к объекту, для которого она вызвана через указатель this.

Рассмотрим пример:

comp &operator+(comp)

{real=real+x.real;

im=im+x.im;

return \*this;}

В примере реализована для класса comp перегрузка операции сложения. Здесь последовательно складываются действительные и мнимые части. Возвращение результата происходит через указатель this. Если нужно вернуть адрес объекта, то пишется return this.

7. Друзья

Дружественной функцией класса называется функция, которая, не являясь его компонентом, имеет доступ к его защищенным и собственным компонентам.

Функция не может стать другом класса “без его согласия”. Для получения прав друга функция должна быть описана в теле класса со спецификатором friend. Именно при наличии такого описания класс предоставляет функции права доступа к защищенным и собственным компонентам.

class C

{ …

friend class A; }

Все функции класса A имеют доступ к закрытым полям класса C.

Дружба не носит ”сквозного” характера (не обладает свойством транзитивности): если класс A друг класса B, а класс B друг класса C, то это не означает, что A друг C.

Отметим особенности дружественных функций. Дружественная функция при вызове не получает указателя this. Объекты классов должны передаваться дружественной функции только явно через аппарат параметров. При вызове дружественной функции нельзя использовать операции выбора:

имя\_объекта.имя\_функции и указатель\_на\_объект->имя\_функции

### 8. Перегрузка операций

На все операции языка C++, кроме операций объявления, new, delete, и других операций, связанных с определением производных типов данных, распространяется свойство полиморфизма, т.е. возможности использования в различных случаях для одной и той же операции операндов различных типов. Так, например, операция сложения позволяет “смешивать” типы int, double, float и другие в одном выражении. Такой полиморфизм обеспечен внутренними механизмами языка C++.

Таким образом, нельзя перегружать такие операции: **.** **::** **\*** **?:**

Чтобы появилась возможность использовать стандартную для языка C++ операцию с необычными для нее данными, необходимо специальным образом определить ее новое поведение. Это возможно, если хотя бы один из операндов является объектом некоторого класса, т.е. введенного пользователем типа. В этом случае применяется механизм, во многом схожий с механизмом определения функций. Для распространения действия операции на новые пользовательские типы данных программист определяет специальную функцию, называемую “операция-функция” (operator function). Формат определения операции-функции:

тип\_возвращаемого\_значения operator знак\_операции (спецификация параметров операции-функции) {операторы тела операции-функции}

При необходимости может добавляться и прототип операции-функции с таким форматом:

тип\_возвращаемого\_значения operator знак\_операции (спецификация параметров операции-функции);

И в прототипе, и в заголовке определения операции-функции используется ключевое слово operator, вслед за которым помещен знак операции. Если принять, что

конструкция operator знак\_операцииесть имя некоторой функции,то определение и прототип операции-функции подобны определению и прототипу обычной функции языка C++. Например, для распространения действия бинарной операции \* на объекты класса T может быть введена функция с заголовком T operator \*(T x, T y).

Определенная таким образом операция (в нашем примере операция “ звездочка”) называется перегруженной (по-английски - overload), а сам механизм – перегрузкой или расширением действия стандартных операций языка C++.

Количество параметров у операции-функции зависит от арности операции и от способа определения функции. Операция-функция определяет алгоритм выполнения перегруженной операции, когда эта операция применяется к объектам класса, для которого операция-функция введена. Чтобы явная связь с классом была обеспечена, операция-функция должна быть либо компонентом класса, либо она должна быть определена в классе как дружественная, либо у нее должен быть хотя бы один параметр типа класс (или ссылка на класс).

Если для класса T введена операция-функция с приведенным выше заголовком и определены два объекта A и B класса T,то выражение A\*B интерпретируется как вызов функции operator \* (A,B).

Рассмотрим пример. Реализуем перегрузку операции сложения для класса комплексных чисел.

class comp

{float im; float real;

public:

comp(float i, float r)

{real=r;

im=i;}

comp operator +(comp X)

{return comp(im+X.im, real+X.real);}

}

void main()

{ **…**

comp C1(1,1), C2(5,5),C3;

C3=C1.operator+(C2) // Прямой вызов операции-функции. Не используется.

C3=C1+C2 // Косвенный вызов операции-функции.

**…**

}

Компилятор по типам объектов С1 и С2 определяет, что необходимо реализовать не просто сложение двух скаляров, как это бывает в обычном использовании операции +, а вызвать перегруженную функцию operator +.Так как при определении класса поля im и real доступны функциям класса, есть необходимость определять только второй объект (X в нашем примере).

В языке C++ требуется, чтобы операции присваивания, индексации и косвенного обращения к полям класса (->) обязательно определялись как методы, т.е. как функции-члены класса.

Когда левый операнд операции является представителем класса, перегруженную операцию нужно определять как метод этого класса.

Для многих операций C++ существуют свои особенности при перегрузке (доопределении). Так, унарные операции переопределяются с описанием операции-функции без аргумента, например:

class A

{ **…**

A operator --() {текст функции}

**…** }

Соответственно доопределение бинарной операции использует описание операции-функции с одним аргументом, т.к. вторым является объект, для которого вызвана операция. Следует также помнить, что операция присваивания “=” может перегружаться только объявлением метода без описателя static. То же относится к операциям “()” и ”[]”.

Посмотрим, как будет выглядеть перегрузка операции присваивания для примера с комплексными числами.

comp & operator =([const] comp & X)

{real=X.real;

im=X.im;

return \*this;}

Если указываем const, то это показывает, что параметр не должен изменяться внутри функции, а кроме того, позволяет обрабатывать константные объекты.

Операция присваивания не наследуется.

**Константные объекты и константные методы**

const Loc NK(0,0); //константный объект

После инициализации попытки изменения константного объекта отслеживаются и пресекаются компилятором.

Объявление константной функции в теле класса выглядит следующим образом:

Прототип const;

ПРАВИЛО:

Константные методы

1. не должны менять значения элементов класса;
2. не должны вызывать другие неконстантные методы класса.

Константные методы могут применяться как для константных, так и для неконстантных объектов.

Литература

1. М.Уэйт, С.Прата, Д.Мартин Язык Си: Пер с англ.-М.: Мир, 1988.-463 с.,ил.
2. Уинер Р. Язык Турбо Си: Пер с англ.-М.: Мир, 1991.-384 с.,ил.
3. Берри Р., Микинз Б. Язык Си: введение для программистов: Пер. с англ.-М.:Финансы и статистика, 1988.-с.,ил.
4. TURBO C++. Borland International. Inc. 1990.