МИНИСТЕРСТВО ОБРАЗОВАНИЯ РФ
МИНИСТЕРСТВО НАУКИ И ТЕХНОЛОГИИ РФ
МОСКОВСКИЙ КОМИТЕТ ОБРАЗОВАНИЯ
КОМИТЕТ ПО ДЕЛАМ СЕМЬИ И МОЛОДЕЖИ ПРАВИТЕЛЬСТВА МОСКВЫ

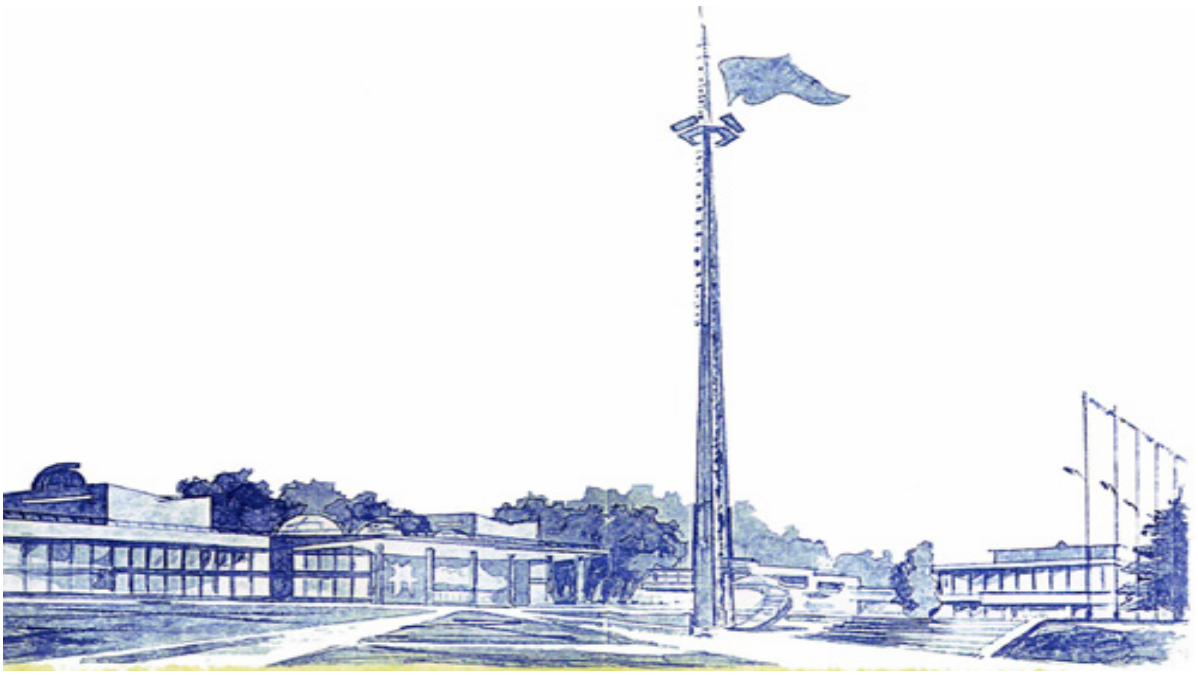
МОСКОВСКИЙ ГОРОДСКОЙ ДВОРЕЦ ДЕТСКОГО (ЮНОШЕСКОГО) ТВОРЧЕСТВА

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНСТИТУТ РАДИОТЕХНИКИ, ЭЛЕКТРОНИКИ И АВТОМАТИКИ (ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)

Булицын С.А., Кузнецова Ю.В., Малаханов Д.Е. и др.

Под редакцией А.Ю. Паршина

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



Москва 2002/2003

УДК 681.3.06
ББК 32.88-421

Главный редактор: Первый зам. директора МГДД(Ю)Т В.Е.Соболев
Рук. эксп. техн. комплекса: В.И.Минаков
Литературный редактор: Л.А.Карась
Технологическое обеспечение: В.Т.Матчин, С.В.Свечников, А.А.Савочкин
Выпускающий редактор: С.В. Свечников
Корректор и макетирование: Д.А. Блинников

Богомолов А., Булицын С., Горохова Т., Гуцалюк Е., Домахин Д., Киселева М., Котляров Ю., Кошеляев Л., Кудряшов А., Кузнецова Ю.В., Малаханов Д., Матчина Н., Ригина Е., Селезнева М., Шалаева А. Объектно-ориентированное программирование / Под редакцией А.Ю. Паршина МГДД(Ю)Т, МИРЭА, М., 2002. с.31

Объектно-ориентированное программирование рассматривается на примере современного языках программирования С и С++. В языках С/С++ есть ряд операторов которых раньше не было в других языках программирования. В их числе побитовые операторы, операторы инкрементирования и декрементирования, условный оператор, оператор тока, оператор запятая и др.

Здесь же рассматриваются основные принципы объектно-ориентированного программирования: Инкапсуляция - Класс рассматривается как единство свойств и действий. Инкапсуляцией называется объединение в абстрактном классе его статических и динамических свойств, при этом, когда рассматриваем класс с точки зрения программирования, то свойства класса - переменные классы, а это статика класса. Действия, которые совершаются над классами, называются функцией класса, а это динамика класса. Наследование - свойство класса создавать себе подобные классы. Исходный класс – производный класс. Производный класс формируется следующим образом:

Рекомендовано для использования по дисциплине «Информатика» (в системе дополнительного образования сектора ИВТ МГДД(Ю)Т и по программам ВШ РФ кафедры ТИССУ МИРЭА).

ISBN 5-8094-0018-3 ББК 3288-421

Лицензия на издательскую деятельность: ЛР №040686 от 27 мая 1994

Адрес в МГДД(Ю)Т: email – cnit@mgdtd.ac.ru 119991, Москва, ул. Косыгина, д.17, комн. 4-21, 4-31.
Адрес в МИРЭА: email – cnit@mirea.ac.ru 117454, Москва, пр-т Вернадского, д. 78.

МГДД(Ю)Т Заказ Тираж

Структура программы на языке процедурного программирования

Языки программирования:

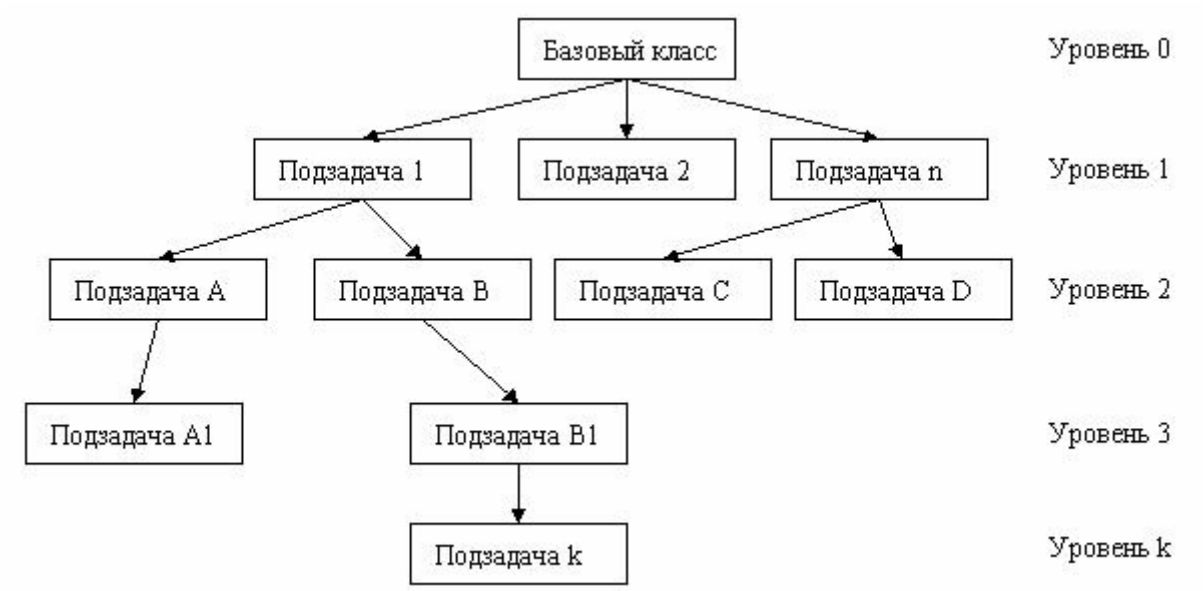
- Процедурные языки;
- Объектно-ориентированные языки.

Процедурные языки характеризуются жестким алгоритмом, который заранее продумывается, созданием программного обеспечения, базируются на аппарате функции.

Структура программы на языке процедурного программирования состоит из четырех основных пункта:

1. Глобальная задача – сложная задача;
2. Глобальная задача разбивается на подзадачи;
3. Продолжается разбиение подзадач на более простые;
4. Процесс разбиения продолжается до тех пор, пока это необходимо.

Отдельные полученные задачи записываются как функции. В результате получаем иерархию функций. Иерархия функций состоит из уровней:



Количество уровней зависит от задачи. Чем ниже уровень, тем легче задача. На самом низком уровне стоит самая простая функция. Функции взаимосвязаны, они передают друг другу параметры. Также возможна передача параметров через глобальную область *(глобальные переменные – это математические константы)*. При изменении функции меняется список параметров. Тогда возникает недостаток процедурного языка – ограничение объема программного кода.

Структура программы на объектно-ориентированном языке

Структура программы на объектно-ориентированном языке состоит из трех пунктов:

1. В основе лежит базовый класс *(класс – это абстрактный тип данных)* – он самый простой;

2. Классы могут быть независимыми;
3. строится иерархия наследования, связь классов, порождающиеся классы является более сложными.

В результате имеем следующую иерархию:

Базовый класс является простейшим из всех классов. Базовых классов может быть несколько и их можно добавлять в процессе эксплуатации. Сложность класса увеличивается с номером уровня. Внизу иерархии стоят самые сложные функции.

Преимущество объектно-ориентированных языков

Объектно-ориентированные языки позволяют создавать:

- Можно дополнять свойства уже существующего класса, при этом менять класс не надо;
- Поставив данный класс в нужное место иерархии, остальные классы тоже не меняют. Создание нового класса;
- Объектный код позволяет увеличить объем поля и написать сотни тысяч слов.

Класс – это специальный абстрактный тип данных, позволяющий хранить в себе свойства и действия.

Свойства:

- Статика (обычно не меняется);
- Динамика (меняется).

Файл – абстрактная сущность

Статика:

имя,
размер.

\+ *класс*

Динамика:

читать из файла,
писать в файл.

Пример:

Дата – абстракция

Статика:

число,
месяц,
год.

\+ *класс*

Динамика:

помнить следующую дату,
помнить предыдущую дату,
вычисление промежутков между датами.

Аналогия – параллель со стандартными типами данных

Стандартный тип данных поддерживается аппаратом.

Основные стандартные типы:

- Целый (2б int);
- Действительный (4б char);
- Символьный (1б float).

Динамика типа – это тот набор операций, который можно выполнять над типом:

1. Целые:

арифметика, логические операции, операции сравнения, функция.	целый результат

2. Символьные:

арифметика, логические операции, операции отношения, свои функции.	символьный результат

По аналогии с основными типами, можно создать абстрактные типы данных.
Действия:

1. Дать имя типу;
2. Определить набор операций, которые можно производить над ними;
3. Объединить в единое целое статические и динамические свойства.

Принято называть статическими свойствами класса – переменными, а его динамические свойства – функциями.

Класс – абстрактный тип данных – сущность.

Статика (свойства):

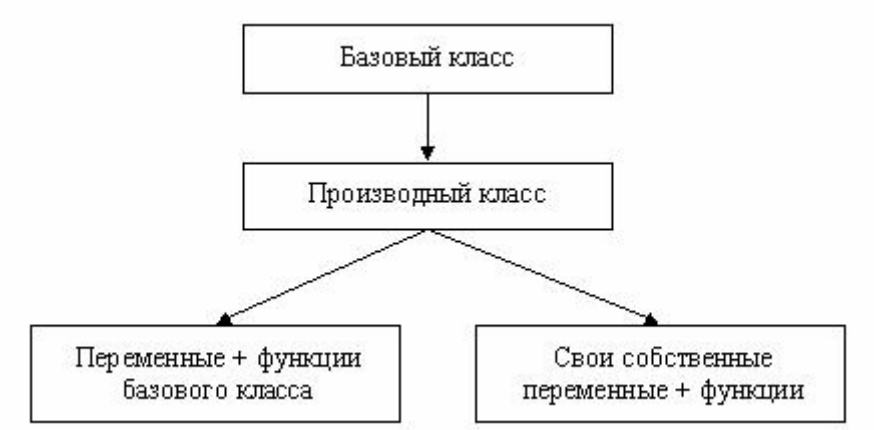
Переменные класса.

Динамика (действия):

Функции класса.

Основные принципы ООП

1. **Инкапсуляция** - Класс рассматривается как единство свойств и действий.
Инкапсуляцией называется объединение в абстрактном классе его статических и динамических свойств, при этом, когда рассматриваем класс с точки зрения программирования, то свойства класса - переменные классы, а это статика класса. Действия, которые совершаются над классами, называются функцией класса, а это динамика класса.
2. **Наследование** - свойство класса создавать себе подобные классы.
Исходный класс – производный класс. Производный класс формируется следующим образом:
1) В него входят все переменные и функции базового класса;
2) Добавляются новые переменные и функции, которых нет в базовом классе.



При этом производном классе переменные и функции базового класса не повторяются, но пользоваться ими можно в производном классе. Производный класс может быть несколько в зависимости от сложности задачи.
3) Полиморфизм (много образов) – это свойство функции класса менять свою сущность в зависимости от внешних воздействий. В применении к программированию это означает, что функция с одним и тем же именем может выполнять различные действия в разных частях программы.

1. Примером полиморфной функции служит функция, имеющая одно и то же имя и разный список параметров.
2. Работа визуальных оболочек.

Уровни защищенности переменных и функций класса

1. Внутренние (private) – использование только самим классом;
2. Внешние (public) – использование самим классом и внешней средой.

Имеется следующий синтаксис класса с уровнями защищенности:

Private:

Внутренняя часть ⎧ декларация внутренних переменных класса
⎩ прототипы внутренних функций.

Public:

Внешняя часть ⎧ декларация внешних переменных класса
⎩ прототипы внешних функций.

Любая из частей public и private может отсутствовать.

Если отсутствует private – класс является полностью не защищенным от внешних воздействий.

Если отсутствует public – класс нельзя использовать, т.к. отсутствуют функции взаимодействия с внешней средой.

Для корректного определения класса закрытыми делаются переменные класса. Это делается для того, чтобы их трудно было испортить непрофессиональному пользователю.

Пример простейшего класса данных:

```
Class date
        {private:int,day,year}
         public: int, input (int,char,int);
int output (int, char*, int);
int sum1 (int,char*, int);
int sum2 (int, char*, int);
int min1 (int, Char*,int);
int min n (int,char*, int);
int koi (int, char*,int,int,char*,int,int)
```

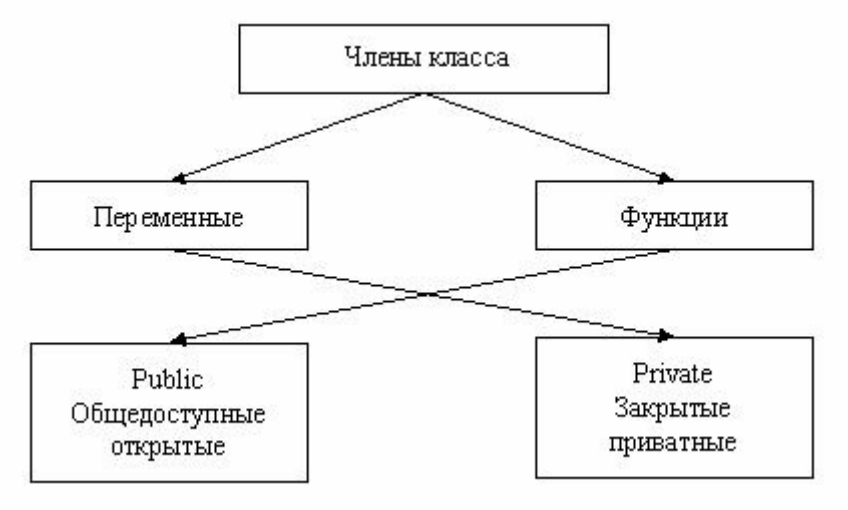
Общая структура программ для ОО кода

1. Подключение библиотек include
2. Описание класса идет как описание типа
3. Описание всех функций
4. Точка когда main() программа

Понятие объекта

Объектом называется переменная типа класса (переменная абстрактно-пользовательского типа).

Синтаксис декларации объектов аналогичен базовому типу.



Операторы объектно-ориентированного программирования

В языках C/C++ есть ряд операторов которых раньше не было в других языках программирования. В их числе побитовые операторы, операторы инкрементирования и декрементирования, условный оператор, оператор тока, оператор запятая и др.

Побитовые операторы

Побитовые операторы обращаются с переменными как с наборами битов, а не как с числами. Эти операторы используются в тех случаях, когда необходимо получить доступ к отдельным битам данных, например при выводе графических изображений на экран. Побитовые операторы могут выполнять действия только над целочисленными значениями. В отличие от логических операторов, с их помощью сравниваются не два числа целиком, а отдельные их биты. Существует три основных вида побитовых операторов: И (&) ИЛИ (|) и исключающее ИЛИ (^). Сюда можно также отнести унарный оператор побитового отрицания (~), который инвертирует значения битов числа.

Операторы сдвига

Языки C/C++ содержат два операторы сдвига: сдвиг влево (<<) и сдвиг вправо (>>). Первый сдвигает битовое представление целочисленной переменной, указанной слева от оператора, влево на количество битов, указанное справа от оператора. При этом освобождающиееся младшие биты заполняются нулями, а соответствующее количество старших битов теряется.

Сдвиг беззнакового числа на одну позицию влево с заполнением младшего разряда нулём эквивалентен умножению числа 2:

```
Unsigned int value = 65 ;    // младший байт: 0100 0001
Value <<= 1 ;                // младший байт: 1000 0010
Cout << value;               // будет выведено 130
```

Инкрементирование и декрементирование.

Операции увеличения или уменьшения значения переменной на 1 очень часто встречаются в программах, поэтому разработчики языка C предусмотрели для этих

целей специальные операторы инкрементирования (++) и декрементирования (--). Эти операторы можно применять только к переменным, но не к константам.

Оператор "запятая"

Оператор запятая (,) позволяет последовательно выполнить два выражения, записанных в одной строке. Результатом является значение выражения, расположенного справа от запятой. Синтаксис оператора следующий:

Левое_выражение, правое_выражение

Чаще всего этот оператор применяется в циклах for, когда условие цикла нужно включить проверку значений нескольких переменных.

Оператор доступа (.)

переменная_типа_класс.член_класса ;

Доступ по этому оператору извне возможен только к октрытому классу public. Под "извне" понимается внешняя функция для класса

Оператор видимости (::)

Назначение оператора – определить к какому классу относиться конкретная функция.

Синтаксис:

Тип имя_класса :: имя_функции (список_параметров_с_указанием_типа)
{тело функции
}

Оператор видимости трансформирует имя_функции в имя_класса + имя_функции.

Оператор ввода/вывода

Ввод-вывод в C++ значительно усовершенствован упрощен благодаря универсальным операторам >> (ввод) и << (вывод). Их универсальность стала возможной благодаря появившемуся в C++ понятию перезагрузки операторов, которое заключается в создании функций, имена которых совпадают с именами стандартных операторов языка. Компилятор различает вызов настоящего и "функционального" операторов на основании типов передаваемых им операндов. Операторы ввода-вывода перегружены так, чтобы поддержать все стандартные типы данных, включая классы.

Операция стрелка → доступа к членам класса

Используется, если объект объявлен как указатель на класс.

y *obj;
obj input ( ); эквивалентно →
(*obj). input ( );

Синтаксис оператора "стрелка".
адрес_объекта → член_класса;
при объявлении объекта: имя_класса*имя_объекта.

Конструктор копирования

Назначение: создавать побитовую копию объекта в следующих ситуациях:

- передача объекта в функцию по значению;
- при возврате объекта из функции через тип функции;
- декларация объекта с использованием инициализации уже существующих объектов;

```
my m1;
my m2 = m1;
```

m1 – новый объект; m2 – уже существующий; то есть требуется сделать копию, переписать значения из m1 в m2. Это делает конструктор копирования.

Синтаксис конструктора копирования.

```
имя_класса (имя класса & obj)
      {
  } тело конструктора
```

Для класса my можно рекомендовать следующий конструктор:

```
my :: my (my & obj)
          {cont << end l << "конструктор копирования";
        x = obj. x;
        y = obj. y; }прямое побитовое копирование
```

Передача объекта из функции через тип функции (т.е. использование return)

Возьмем для класса my функцию суммирования:

Сумма объекта my m_1, m_2;
получить третий объект $m_3 = m_1 + m_2$
то есть, требуется m3. x = m1. x + m2. x
m3. y = m1. y + m2. y

так как x, y – закрытые члены класса, операцию подобного вида во внешней функции выполнить невозможно, поэтому в правой части использовать функцию типа ret (), а в левой – set ().

Функция суммирования объектов имеет вид:

```
 my sum (my m1, my m2)
{ my m3;
  m3. set x (m1. ret x ( ) + m2. ret x ( ));
  m3. set y (m1. ret y ( ) + m2. ret y ( ));
  return (m3);
}
```

Для функции суммирования sum;
m_1, m_2 - передача по значению два раза будет вызван конструктор копирования.
Для m3 будет вызван конструктор по умолчанию.

При возврате m3 из функции будет вызван конструктор копирования, чтобы побитово вернуть созданный внутри функции объект.

Объект m3 является локальной переменной для функции sum ( ),

В процедурном программировании вернуть локальную переменную такого типа из функции невозможно. В ООП подобная операция совершается при помощи конструктора копирования.

Указатель this

Используется только в функциях членах класса. Указатель возвращает объект (адрес объекта), для которого функция применяется.

Пример:

Добавляем в класс my функцию next (), которую вычисляет следующий объект делает

```
x++;
y++;

class my
        {int x, y;
         public:
           input ( ) {… }
                output ( ) {… }
           set…
           ret…
         my* next ( ) {x++; y++; return this;}
               my (my & obj) {… }
          конструкторы…
                деструкторы…
        };
void main ( )
    {my m1, *m2;        - конструктор по  умолчанию
          m1. input ( );
          m2. input ( );

     my m3 = m1;
     my m4 = *m2;     - вызов конструктора копирования
```

Получить следующий объект

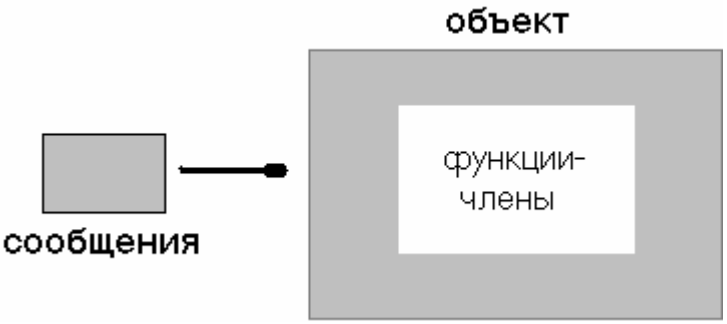
```
     m1. next ( );        - без this
     my m5 = m1. next ( );  копирование идет из указателя this.
```

Классы

Понятие класс (class) относится ко всем объектам, которые ведут себя одинаково. Например, все окружности имеют вполне определенную форму, они обладают такими атрибутами, как местоположение, цвет, диаметр. Объект – это конкретный экземпляр данного класса. Например, Земля имеет размер, цвет и местоположение, отлич-

ные от аналогичных параметров для Луны или Солнца. Связь между классом и объектами в сущности такая же, как между типом и переменными этого типа.

Каждый класс объектов может реагировать на строго определенные сообщения. Так происходит потому, что каждый класс обладает набором функций, которые связаны с объектами класса. Функции являются частью этого класса объектов – его членами. На рисунке показан объект, содержащий функции-члены. Программа посылает этому объекту сообщения (messages), которые вызывают функции-члены (member functions) данного объекта. Затем эти функции-члены обрабатывают объект.



Эти функции называются функциями-членами, поскольку принадлежат классу, то есть являются его членами. Функции-члены программируются так же, как обычные функции, однако объявляются в классе и могут использоваться только с объектами этого класса.

Конструкторы

Допустим, имеется объект класса Clock. При объявлении этого объекта, он автоматически инициализуется. Это означает, что при создании нового объекта класса Clock переменной timestarted присваивается текущее системное время. Кто (или вернее что) это делает?

Для этого нужно определить специальную функцию, которая будет специально вызываться при создании каждого объекта. В языке C++ это можно сделать при помощи специальной функции, которая называется конструктором (constructor).
Конструктор похож на любую другую функцию-член, за исключением следующего:

1. Имя конструктора совпадает с именем класса. Например, конструктором класса Clock является функция Clock().
2. При создании нового объекта конструктор вызывается автоматически. Например, если создать два объекта mine и yours класса Clock, то конструктор Clock() будет вызван дважды- один раз при создании объекта mine и другой при создании объекта yours.
3. Конструктор нельзя вызвать из программы напрямую. Например, нельзя написать инструкцию mine.Clock(); Конструктор вызывается только однажды – при создании объекта.
4. У конструктора нет возвращаемого типа. Возможно существование нескольких конструкторов с разными списками аргументов.

Простейшие правила проектирования класса

1) Переменные класса находятся в разделе privat.
2) Для каждой переменной класса в классе должна быть функция установки.
3) Функции установки обычно являются открытыми.
4) Для каждой закрытой переменной класса в классе должна быть функция доступа.
5) Функция доступа (обычно) расположена в открытой части класса.

Встроенные функции (in line)

Встроенными функциями называются функции класса, описанные внутри класса, то есть тело функции находится внутри класса. Встроенными могут быть функции, которые не содержат сложных операций if, вложенных в цепи.

Простейший класс:

```
Class my:
    {publik
        int x,y;
     publik:
inline    int funk 1(void){retun(x+y);}
          int funk 2(void){ retun(x*y);}
          void set x(int var){x=var;}
          void set y(int var){y=var;}
          int ret x(void){return x;}
          int ret y(void){return y;}
```

Функция узнается компилятором по двойным фигурным скобкам. Это объясняет их необходимость.

Массив объектов класса

Массив – это набор объектов одного типа. Определение массива включает тип элементов, идентификатор и размер в квадратных скобках. Размер массива должен быть ненулевым положительным целым. Размер массива может быть константным выражением. Элементы массива нумеруются, начиная с нуля.

Синтаксис объявление (декларация) массива объектов:
имя_класса_имя_массива [размер1], [размер2] … [размерn] – n-мерный массив, где размер1, размер2, … размерn – некоторые числовые константы.

Синтаксис доступа к членам класса:
arr[i].член_класса
Доступ к элементам массива объектов типа, определенного пользователем, осуществляется по индексу (номеру) элемента в массиве. Например:
arr[0] – первый элемент
arr[1] – второй элемент
...
arr[размер-1] – последний элемент.

1. Объем памяти для массива объектов

$V = \sum_{i=1}^{n} V_i$, где V_i - объем памяти для i-га элемента и n – размер массива.

2. Указатель на объект

Определить указатель на объект, означает определить адрес начала памяти, с которого расположен наш объект. Указатели определяются по модификатору типа * вместе с другой информацией о типе в объявлении. Этот же символ используется для оператора разыменования, который возвращает объект, на который ссылается указатель.

Синтаксис указателя
имя_класса * имя_объекта
синтаксис обращения к членам класса
(* имя_объекта).член_класса;
3. Способы передачи объекта в функции

Объект для программирования полностью наследует способы передачи параметров функции от процедурного программирования. Всего способов передачи три:

3.1. По назначению:

1. Создается копия, передавшая параметры внутри функции;
2. Внутри функции идет работа с копией (копия области памяти);
3. При выходе из функции копия разрушается, т.е. все изменения, которые были сделаны в копии, пропадают.

```
void func(int x); - прототип
func(x); - вызов
```

3.2. По адресу:

1. Копия параметра внутри функции не создается;
2. Работа внутри функции с оригиналом параметра;
3. Все изменения, которые происходят внутри функции, сохраняются при выходе из нее.

```
void func(int*x);
func(&x);
```

3.3. По ссылке:

При передаче параметра по ссылке, технология передачи параметра аналогично передачи по адресу, но при этом используется ссылочный тип.

```
void func(int&x);
func(x);
```

4. Передача параметров в функцию применительно к объектам

При передаче объекта в функцию по значению требуется создать копию объекта, а при выходе из такой функции требуется разрушить копию.

Создание копии:

1. Копию создает конструктор копии, если он есть в классе;
2. Если конструктора копии нет в классе, то копия создается компилятором.

Разрушение копии:

1. Разрушает копию деструктор, если он описан в классе;
2. Если нет деструктора в классе, то компилятор моделирует деструктор.

Объектно-ориентированное программирование обеспечивает возможность многократного использования программного кода. Если нужен класс объектов, очень похожих на существование, но с несколько отличными свойствами, можно построить новый класс из уже существующего. В C++ исходный класс называется базовым (base class), а тот, который создается из него – производным (derived class).

Каждый объект производного класса может одновременно считаться объектом соответствующего базового класса. Например, паровоз остается вагоном, секундомер – часами.

Производные классы обладают очень полезным свойством: все, что уже работает в базовом классе, будет автоматически наследоваться (inherited) производным классом.

Потоковый ввод/вывод

Для вывода данных ваша программа направляет данные в поток. В этом суть системы вывода информации. Все данные, предназначенные для вывода, передаются в специальный объект (cout), который и отображает их на экране.

Если у вас есть целая переменная namber, то вы можете написать ее значение на экране с помощью следующей инструкции:

Cout<<namber;

Аналогично можно вызвать другой объект (cin), который будет собирать данные, набираемые на клавиатуре. Затем данные извлекаются из этого объекта и передаются переменным программы. В этом случае ввести значения переменной namber можно с помощью следующей инструкции:

Cin>>namber;

Имена cin и cout произошли от слов C input (ввод C) и C output (вывод C).

Операции ввода и вывода реализуются с помощью специальных операторов <<и>>. Онинапоминают, чтопри выводе поток направлен от переменной к объекту (cout<<namber), а при вводе – от объекта к переменной (cin>>namber).

- потоковый вывод

Синтаксис потокового вывода строится из объекта cout, оператора << и переменной (или константы):

Cout<<идентификатор_переменной;

ПР: cout<<namber;

Если выводятся значения нескольких переменных, то перед каждой из них должен стоять свой оператор вывода (<<):

Cout<”The result is: “<<namber<<” and that is final”;

При выводе информации на экран пробелы между значениями сами по себе не появятся.

Например:

Int namber=32;

Char name[ ]=”Sonny Bonds”;

Cout<<name<<namber;

После выполнения этих трех инструкций на экране появится следующее:

Sonny Bonds32

Две инструкции вывода также не приведут к выводу данных в две строки. Другими словами, в предыдущем примере результат будет тем же, даже если выводить на экран данные с помощью двух инструкций:

Cout<<name;

Cout<<namber;

Пробел

Если требуется вставить пробел между значениями, то проще всего вывести его специально:

Cout<<name<<” “<<namber;

Можно использовать и более подробное описание каждого значения:

Cout<<”Customer Name: “<<name<<” code is; “<<namber;

Новая строка

Имеется возможность выводить данные с новой строки. Для этого существуют два способа: вывести в поток употребляющий символ \n (новая строка) или манипулятор endl.

Создание новой строки с помощью управляющего символа:

Cout<<"Customer Name: "<<name<<\n<<" code is: "<<number;

Создание новой строки с помощью манипулятора:

Cout<<"Customer Name: "<<name<<endl<<" code is: "<<namber;

Оба варианта приведут к появлению на экране одинаковой информации:

Customer Name: Sonny Bonds

Code is: 32

- потоковый ввод

Синтаксис потокового ввода строится из объекта cin, оператора >>и переменной, которой присваивается вводимое значение:

Cin>>имя_переменной;

Следующая инструкция считывает значение с клавиатуры и сохраняет его в переменной i:

Cin>>i;

С помощью одной инструкции ввода можно присвоить значения более чем одной переменной, например:

Cin>>i>>j>>k;

Можно поэкспериментировать со следующей программой, в которой сначала запрашивается ввод трех значений, а затем эти значения выводятся на экран:

```
#include<iostream.h>
#ibclude<iomanip.h>
void main ()
{
      int i, j, k;
      cout<<"Enter three values:\n";
      cin>>i>>j>>k;
      cout<<endl<<"Your values are:"<<\n;
      cout<<i<<j<<k;
}
```

Запустив такую программу, можно убедиться, что введенные значения выводятся правильно.

Можно перейти на новую строку на экране, включив в поток либо манипулятор ввода/вывода endl, либо управляющий символ \n.

Когда управляющий символ будет найден в символьной строке или в потоке вывода, он будет интерпретирован как инструкция перехода на новую строку. Чтобы управляющий символ новой строки оказался в потоке вывода, его либо вставляют в другую строку, либо указывают отдельно. Управляющий символ может заключаться как в двойные, так и в одинарные кавычки. С точки зрения пользователя эффект будет одинаков. Однако двойные кавычки генерируют оканчивающуюся нулем строку, а одинарные кавычки – одиночный символ.

Перегрузка

Говорят, что имя перегружено, если для него задано несколько различных описаний функций в одной области видимости. При использовании имени выбор правильной функции производится путем сопоставления типов формальных параметров с типами фактических параметров, например:

```
double abs(double);
int abs(int);
abs(1); // вызов abs(int)
abs(1.0); // вызов abs(double)
```

Поскольку при любом типе T и для самого T , для и T& допустимо одно и то же множество инициализирующих значений, функции, типы параметров которых различаются только использованием, или не использованием ссылки, не могут иметь одинаковые имена, например:

```
int f(int i)
{
// ...
}

int f(int& r)   // ошибка: типы функций
{               // недостаточно различны
// ...
}
```

Аналогично, поскольку для любом типе T для самого T, const T и volatile T допустимо одно и то же множество инициализирующих значений, функции, типы параметров которых отличаются только указанной спецификацией, не могут иметь одинаковые имена. Однако, различить const T&, volatile T& и просто T& можно, поэтому допустимы определения функций с одним именем, которые различаются только в указанном отношении. Аналогично, допустимы определения функций с одним именем, типы параметров которых различаются только как типы вида const T*, volatile T* и просто T*. Не могут иметь одинаковые имена функции, которые отличаются только типом возвращаемого значения. Не могут иметь одинаковые имена функции-члены, одна из которых статическая, а другая нет. С помощью конструкции typedef не создаются новые типы, а только определяется синоним типа, поэтому функции, которые отличаются только за счет использования типов, определенных с помощью typedef, не могут иметь одинаковые имена. Приведем пример:

```
typedef int Int;
void f(int i) { /* ... */ }
void f(Int i) { /* ... */ } // ошибка: переопределение f
```

С другой стороны все перечисления считаются разными типами, и с их помощью можно различить перегруженные функции, например:

```
enum E { a };
void f(int i) { /* ... */ }
void f(E i) { /* ... */ }
```

Типы параметров, которые различаются только тем, что в одном используется указатель *, а в другом массив [], считаются идентичными. Напомним, что для типа параметра важны только второй и последующие индексы многомерного массива. Подтвердим сказанное примером:

```
f(char*);
```

```
f(char[]);      // идентично f(char*);
f(char[7]);     // идентично f(char*);
f(char[9]);     // идентично f(char*);
g(char(*)[10]);
g(char[5][10]); // идентично g(char(*)[10]);
g(char[7][10]); // идентично g(char(*)[10]);
g(char(*)[20]); // отлично от g(char(*)[10]);
```

Сопоставление описаний

Два описания функций с одинаковыми именами относятся к одной и той же функции, если они находятся в одной области видимости и имеют идентичные типы параметров. Функция-член производного класса относится к иной области видимости, чем функция-член базового класса с тем же именем. Рассмотрим пример:

```
class B {
public:
int f(int);
};
class D : public B {
public:
int f(char*);
};
Здесь D::f(char*) скорее скрывает B::f(int), чем перегружает эту функцию.
void h(D* pd)
{
pd->f(1);       // ошибка: D::f(char*) скрывает B::f(int)
pd->B::f(1);  // нормально
pd->f("Ben");          // нормально, вызов D::f
}
```

Функция, описанная локально, находится в иной области видимости, чем функция с файловой областью видимости.

```
int f(char*);
void g()
{
extern f(int);
f("asdf");      // ошибка: f(int) скрывает f(char*) поэтому
// в текущей области видимости нет f(char*)
}
```

Для разных вариантов перегруженной функции-члена можно задать разные правила доступа, например:

```
class buffer {
private:
char* p;
int size;
protected:
buffer(int s, char* store) { size = s; p = store; }
// ...
public:
buffer(int s) { p = new char[size = s]; }
```

```
};
```

Адрес перегруженной функции

Когда функция с некоторым именем используется без параметров, среди всех функций с таким именем в текущей области видимости выбирается единственная, которая точно соответствует назначению. Назначением может быть: инициализируемый объект; левая часть операции присваивания; формальный параметр функции; формальный параметр пользовательской операции; тип значения, возвращаемого функцией. Отметим, что если f() и g() являются перегруженными функциями, то для правильной интерпретации f(&g) или эквивалентного выражения f(g) нужно рассмотреть пересечение множеств выбора для f() и g(). Приведем пример:

```
int f(double);
int f(int);
int (*pfd)(double) = &f;
int (*pfi)(int) = &f;
int (*pfe)(...) = &f; // ошибка: несоответствие типов
```

Последняя инициализация ошибочна, не из-за неоднозначности, а потому, что не определено ни одной функции f() типа int(...). Отметим, что не существует никакого стандартного преобразования указателя на функцию одного типа в указатель на функцию другого типа. В частности, даже если B является общим базовым классом D, две следующие инициализации недопустимы:

```
D* f();
B* (*p1)() = &f; // ошибка void g(D*);
void (*p2)(B*) = &g; // ошибка
```

Перегруженные операции

Перегружать можно большинство операций.

имя-функции-оператор:

operator операция

операция: один из

new delete + - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= && || ++ -- , ->* -> () []

Две последние операции - это вызов функции и индексация. Можно перегружать следующие (как бинарные, так и унарные) операции:

```
+ - * &
```

Нельзя перегружать следующие операции:

```
. .* :: ?: sizeof
```

а также и специальные символы препроцессора # и ##. Обычно функции, задающие операции (функция-оператор) не вызываются явно, к ним обращаются для выполнения операций. Однако, к ним можно обращаться явно, например:

```
complex z = a.operator+(b); // complex z = a+b
void* p = operator new(sizeof(int)*n);
```

Функция-оператор может быть функцией-членом или иметь по крайней мере один параметр типа класс или ссылка на класс. Нельзя изменить приоритет, порядок выполнения или число операндов операции, но можно изменить предопределенное назначение таких операций: =, унарная & и ,(запятой), если они применяются к объекту типа класс. За исключением функции operator=(), функция-оператор наследуется. Эквивалентность некоторых операций над основными типами (например, ++a эквивалентно

a+=1) может не сохраняться для таких же операций над классами. Для некоторых операций требуется, чтобы в случае использования основных типов операнд был адресом (например, для +=). Это требование может быть снято, если операция задана над классами. Перегруженная операция не может иметь стандартные значения параметров.

Унарные операции

Префиксную унарную операцию можно задать с помощью нестатической функции-члена, без параметров или с помощью функции, не являющейся членом, с одним параметром. Таким образом, для всякой префиксной унарной операции @, выражение @x может интерпретироваться как x.operator@() или как operator@(x). Если описаны функции-операторы обоих видов, то какая из них будет использоваться при вызове, определяется правилами сопоставления параметров. Постфиксные унарные операции, такие как ++ и -- , объясняются в соответствующем разделе.

Бинарные операции

Бинарную операцию можно задать с помощью нестатической функции-члена, имеющей один параметр, или с помощью функции, не являющейся членом, с двумя параметрами. Таким образом, для всякой бинарной операции @ выражение x@y может интерпретироваться как x.operator@(y) или как operator@(x,y). Если описаны функции-операторы обоих видов, то какая из них будет использоваться при вызове, определяется правилами сопоставления параметров.

Присваивания

Функция присваивания operator=() должна быть нестатической функцией-членом. Она не наследуется. Более того, если пользователь не определил для класса X функцию operator=, то используется стандартная функция operator=, которая определяется как присваивание по членам для класса X.

```
X& X::operator=(const X& from)
{
// копирование по членам X
}
```

Вызов функции

Вызов функции есть конструкция вида:

первичное-выражение (список-выражений opt)

Она считается бинарной операцией, в которой первичное-выражение представляет первый операнд, а список-выражений (возможно пустой), - второй операнд. Именем, задающим функцию, служит operator(), и вызов x(arg1,arg2,arg3) для объекта класса x интерпретируется как x.operator()(arg1,arg2,arg3). Функция operator() должна быть нестатической функцией-членом класса x.

Индексация

Индексация, определяемая как:

первичное-выражение [выражение]

считается бинарной операцией. Выражение с индексацией x[y] для объекта класса x интерпретируется как x.operator[](y). Функция operator[] должна быть нестатической функцией-членом класса x.

Доступ к члену класса

Доступ к члену класса определяется с помощью операции ->:

первичное-выражение -> первичное-выражение

Он считается унарной операцией. Для объекта класса x выражение x->m интерпретируется как (x.operator->())->m. Отсюда следует, что функция operator->() должна возвращать или указатель на класс, или ссылку на класс, или объект класса, для которого определена функция operator->(). Она должна быть нестатической функцией-членом класса.

Инкремент и декремент

Функция с именем operator++ и с одним параметром задает для объектов некоторого класса операцию префиксного инкремента ++. Функция с именем operator++ и с двумя параметрами задает для объектов некоторого класса операцию постфиксного инкремента ++. Для постфиксной операции ++ второй параметр должен быть типа int, и, когда в выражении встречается операция постфиксного инкремента, функция operator++ вызывается со вторым параметром, равным нулю.

Префиксные и постфиксные операции декремента -- определяются аналогичным образом.

Дружественные функции

Ключевое слово – Friend

Называется функция внешне к классу, которая имеет доступ к закрытым переменным класса.

Синтаксис:

Friend тип имя _ функции (пар-ры);

Ключевым словом friend дружественная функция описывается внутри класса, но вне класса описывается тело функции без ключевого слова friend.

Случаи использования friend функции

1. Если нужна внешняя функция для класса, имеющая доступ к внутренним переменным класса.

Функции сортировки можно сделать в friend функции.

2. Несколько классов и нужная функция, имеющая доступ к закрытым переменным классов.
3. Если необходимо перегрузить левосторонние операции

   My obj;

   My obj 1 = obj+5;

Описанный в классе my operator + не может выполнить следующую операцию:

My obj 2 = 5 + obj; хотим к числу прибавить объект, по определению перегруженный. Оператор левостороннюю операцию для объекта выполнить не может, поэтому в класс добавляют аналогичные friend операторы, так как friend функция может иметь любое кол-во параметров. Следовательно, friend оператор может иметь аналогично любое кол-во параметров.

Чтобы прибавить слева число (наш случай) необходимо два параметра:

1) int
2) тип my.

Применение :
Void main ()
My m1(1,2), m2(10,20);
My m3 = 1+m1 ;
My m4 = m2+2;

Сложные выражения с объектами

Под сложным выражением понимают множество арифметических операций, выполненных в одном операторе.
Пусть : в сл. My перегружен + - * /
Тогда можно писать следующее выражение:
My m1,m2,m3;
My obj = m1+m2*m3-5m3+6*m4-6*(m1/m3)*m2;
Такие выражения нормально работают, если операторы перегружены таким образом, что возвращают ссылку на тип.
Для класса матриц обязательная демонстрация выражения с матрицами.
Ссылка на тип
Тип & operator @ (…)

Перегрузка потоковых операций ввода/вывода

Cin » имя_ переменной; - ввод переменной из потока cin
Cin – стандартны поток ввода (клавиатура, консоль на входе)
Cout « имя _переменной; - вывод в поток
Cout – стандартный поток вывода (монитор, консоль на выходе)

В операциях ввода/вывода участвуют в перегрузке операции побитового сдвига влево и вправо » и «.

Перегруженный оператор ввода возвращает ссылку на поток iostream, поэтому в 1-ом операторе можно вводить несколько переменных.

Оператор ввода определен для всех базовых типов и модификаций (unsigned, short, long).

Процесс ввода/вывода без форматирования происходит в два этапа:
1. компилятор читает переменные и тип, т.е. тип переменной известен.
2. когда идет операция ввода/вывода вызывается нужный перегруженный оператор в соответствии с типом переменной.
Операторы ввода/вывода:
Хотим применить те же самые операторы « и » для объектов абстрактных классов.
Хотим: date d1, d2;
Cin » d1 » d2
Cout « d1 « "…" « d2;
Т.е. хотим применить оператор ввода/вывода вместо функции членов класса input и output.
Для этого требуется перегрузить потоковые операции в абстрактном классе.
1. friend
cin »
cout «
1терм-имя потока стандарт. (в классе ivs), т.е. левым термом не может быть объект абстрактного класса. Поэтому потоковые операции в абстрактном классе должны иметь более одного параметра.

2.friend iostream & operator »
(iostream & in, имя _ класс & имя объекта)
полностью соответствует оператору » из класса iostream
1 параметр – входной поток in – типа iostream &.
3.friend iostream & operator «
(iostream & out, имя_ класса имя объекта);
соответствует оператору вывода из класса iostream.
1параметр – выходной поток out. – типа iostream &.
4.добавить оба оператора в абстрактный класс и вне класса описать как функцию.
» ~ input ( )
« - output ( )
пример:

```
class my
{…
friend iostream & operator »
(iostream & in, my obj);
friend iostream & operator «
(iostream out, my obj);
…
};
```

вне класса

```
iostream operator » (iostream & in, my & obj)
{in » obj.x » obj.y;
return in;
}
```

my & obj – т.к. надо вернуть введенную переменную во внешнюю среду.

```
iostream & operator « (iostream & out, my obj)
{ out « obj.x « “ “ « obj.y;
out « end l ;
return out;
}
void main ( )
{ my m1, m2;
in = cin » m1»m2;
out = cout « m1 « m2;
};
```

Шаблоны типа

С помощью шаблонов типа можно достаточно просто определить и реализовать без потерь в эффективности выполнения программы и, не отказываясь от статического контроля типов, такие контейнерные классы, как списки и ассоциативные массивы. Кроме того, шаблоны типа позволяют определить сразу для целого семейства типов обобщенные (генерические) функции, например, такие, как sort (сортировка). В качестве примера шаблона типов и его связи с другими конструкциями языка приводится семейство списочных классов. Чтобы показать способы получения программы из в значительной степени независимых частей, приводится несколько вариантов шаблонной функции sort(). В конце определяется простой шаблон типа для ассоциативного масси-

ва и показывается на двух небольших демонстрационных программах, как им пользоваться.

Одним из самых полезных видов классов является контейнерный класс, т.е. такой класс, который хранит объекты каких-то других типов. Списки, массивы, ассоциативные массивы и множества - все это контейнерные классы. Но контейнерные классы обладают тем интересным свойством, что тип содержащихся в них объектов не имеет особого значения для создателя контейнера, но для пользователя конкретного контейнера этот тип является существенным. Следовательно, тип содержащихся объектов должен параметром контейнерного класса, и создатель такого класса будет определять его с помощью типа-параметра. Для каждого конкретного контейнера (т.е. объекта контейнерного класса) пользователь будет указывать каким должен быть тип содержащихся в нем объектов.

Использование шаблонных классов означает наличие шаблонных функций-членов. Помимо этого, можно определить глобальные шаблонные функции, т.е. шаблоны типа для функций, не являющихся членами класса. Шаблон типа для функций порождает семейство функций точно также, как шаблон типа для класса порождает семейство классов. Эту возможность мы обсудим на последовательности примеров, в которых приводятся варианты функции сортировки sort(). Каждый из вариантов в последующих разделах будет иллюстрировать общий метод. Как обычно мы сосредоточимся на организации программы, а не на разработке ее алгоритма, поэтому использоваться будет тривиальный алгоритм. Все варианты шаблона типа для sort() нужны для того, чтобы показать возможности языка м полезные приемы программирования. Варианты не упорядочены в соответствии с тем, насколько они хороши. Кроме того, можно обсудить и традиционные варианты без шаблонов типа, в частности, передачу указателя на функцию, производящую сравнение.

Введение операций с помощью параметров шаблонного класса.

Возможны ситуации, когда неявность связи между шаблонной функцией sort() и шаблонным классом Comparator создает трудности. Неявную связь легко упустить из виду и в то же время разобраться в ней может быть непросто. Кроме того, поскольку эта связь "встроена" в функцию sort(), невозможно использовать эту функцию для сортировки векторов одного типа, если операция сравнения рассчитана на другой тип. Поместив функцию sort()в класс, мы можем явно задавать связь с классом Comparator:

```
template class Sort {
public:
  static void sort(Vector&);
};
```

Не хочется повторять тип элемента, и это можно не делать, если использовать typedef в шаблоне Comparator:

```
template class Comparator {
public:
  typedef T T;  // определение Comparator::T
  static int lessthan(T& a, T& b) {
    return a < b;
  }
  // ...
```

```
};
```

В специальном варианте для указателей на строки это определение выглядит так:

```
class Comparator {
public:
  typedef char* T;
  static int lessthan(T a, T b) {
    return strcmp(a,b) < 0;
  }
  // ...
};
```

После этих изменений можно убрать параметр, задающий тип элемента, из класса Sort:

```
template class Sort {
public:
   static void sort(Vector&);
};
```

Теперь можно использовать сортировку так:

```
void f(Vector& vi,
       Vector& vc,
       Vector& vi2,
       Vector& vs)
{
  Sort< int,Comparator >::sort(vi);
  Sort< String,Comparator >:sort(vc);
  Sort< int,Comparator >::sort(vi2);
  Sort< char*,Comparator >::sort(vs);
}
```

и определить функцию sort() следующим образом:

```
template
void Sort::sort(Vector& v)
{
  for (int i=0; i T sqrt(t);

void f(int i, double d, complex z)
{
  complex z1 = sqrt(i);  // sqrt(int)
  complex z2 = sqrt(d);  // sqrt(double)
  complex z3 = sqrt(z);  // sqrt(complex)
  // ...
```

```
}
```

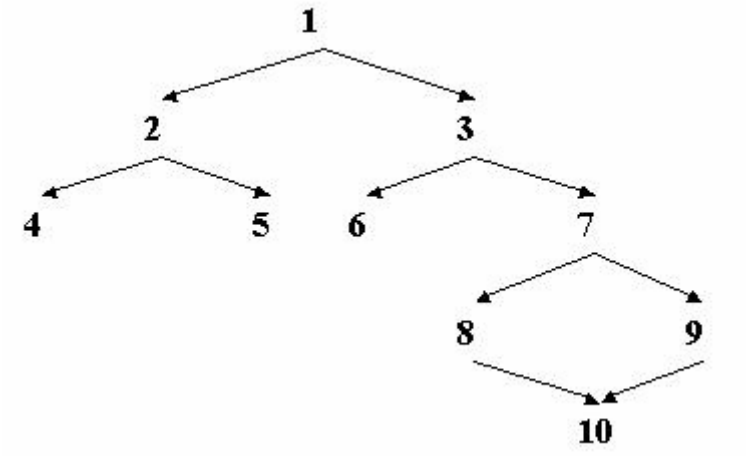
Шаблоны типа дают удобное средство для создания целых семейств классов. Без шаблонов создание таких семейств только с помощью производных классов может быть утомительным занятием, а значит, ведущим к ошибкам. С другой стороны, если отказаться от производных классов и использовать только шаблоны, то появляется множество копий функций-членов шаблонных классов, множество копий описательной части шаблонных классов и во множестве повторяются функции, использующие шаблоны типа.

Тотальное программирование "от класса к классу"

Строгое следование технологии ООП предполагает, что любая функция в программе представляет собой метод для объекта некоторого класса. Это не означает, что нужно вводить в программу какие попало классы ради того, чтобы написать необходимые для работы функции. Наоборот, класс должен формироваться в программе естественным образом, как только в ней возникает необходимость описания новых физических предметов или абстрактных понятий (объектов программирования). С другой стороны, каждый новый шаг в разработке алгоритма также должен представлять собой разработку нового класса на основе уже существующих. В конце концов вся программа в таком виде представляет собой объект некоторого класса с единственным методом run (выполнить). Именно этот переход (а не понятия класса и объекта, как таковые) создает психологический барьер перед программистом, осваивающим технологию ООП.

Программирование "от класса к классу" включает в себя ряд новых понятий. Прежде всего, это – наследование. Новый, или производный класс может быть определен на основе уже имеющегося, или базового. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса. Иначе говоря, новый класс наследует как данные старого класса, так и методы их обработки.

Наследование нужно, для того чтобы расширить уже созданные абстрактные классы новыми свойствами или действиями. Есть несколько правил наследования классов:



1. Создаётся иерархия классов, где классы стоящие ниже по иерархии могут иметь доступ к переменным и функциям выше стоящих классов.
2. Классы стоящие ниже по иерархии- производные классы, относительно классов, которые стоят выше них.(*4,5- производные относительно 2, а 8,9- производные относительно 7)*
3. Классы , которые состоят выше по иерархиям являются базовыми для ниже стоящих классов(*1-базовый для 2 и 3*).
4. Понятие базового и производного класса не предполагают относительность уровня иерархи, т.е. то множество классов, которое стоят на один уровень выше и являются базовыми.
5. Каждый производный класс имеет множество непосредственных родителей, т.е. то множество классов, которые стоят на один уровень выше и являются базовыми.
6. Соответственно: если родитель один- простое наследование, в другом же случаемножественное наследование.
7. Начало иерархии компьютера - это класс, один или более, которые называются протоклассом плюс корень дерева. Обычно бывает 2 или 3 протокласса на практике. Обычно протокол являются пустыми или состоят из пустых виртуальных функций. (Виртуальной называется функция ,сущность которой определяется во время выполнения программы.)
8. Классы стоящие ниже по иерархии имеют дополнительные свойства и функции относительно вышестоящих классов.

Концепция наследования позволяет создавать новые классы, которые используют переменные и функции уже существующих его класса, но не содержит их в своём теле.

Вторым по значимости понятием является полиморфизм. Он основывается на возможности включения в данные объекта также и информации о методах их обработки (в виде указателей на функции). Принципиально важно, что такой объект становится "самодостаточным". Будучи доступным в некоторой точке программы, даже при отсутствии полной информации о его типе, он всегда может корректно вызвать свойственные ему методы. Полиморфной называется функция, определенная в нескольких производных классах и имеющая в них общее имя. Точнее сказать, что полиморфная функция, это группа функций, которая выступает под одним и тем же именем, но в разных классах. Полиморфная функция обладает тем свойством, что при отсутствии полной информации о том, объект какого из производных классов в данный момент обрабатывается, она, тем не менее, корректно вызывается в том виде, который соответствует именно объекту этого класса (Здесь уместен образный термин " многоликая функция"). Практический смысл полиморфизма заключается в том, что программист может сделать регулярным процесс обработки несовместимых объектов различных типов при наличии у них такого полиморфного метода (в Си++ **-виртуальной функции).**

Модули и библиотеки

Модули в С называются *функциями.* Программы на С обычно пишутся путем соединения новых функций, созданных программистом, с функциями, которые поставляются в составе *стандартной библиотеки С.* Стандартная библиотека С предоставляет широкий набор функций для выполнения общих математических вычислений, обработки строк и символов, ввода/вывода и многих других полезных операций. Стандартные функции упрощают работу программиста, поскольку удовлетворяют многим из его

потребностей. Хотя функции стандартной библиотеки технически не являются частью языка С, они неизменно поставляются с системами ANSI C. Обращение к функции осуществляется посредством *вызова функции*. В вызове функции указывается ее имя и передается информация (в качестве аргументов), необходимая функции для выполнения своей задачи. Аналогом такой процедуры служит иерархическая форма управления. Начальник (*вызывающая функция*) просит работника (*вызываемую функцию*) выполнить задание и, когда оно будет выполнено, сообщить об этом.

Каждая стандартная библиотека имеет свой *заголовочный файл,* содержащий прототипы для всех функций данной библиотеки, а также определения различных типов данных и констант, необходимых этим функциям. Программист может создать специализированный заголовочный файл. Определенные программистом заголовочные файлы также должны заканчиваться **.h.** Определенный программистом файл может быть включен директивой препроцессора **#include.** Например, заголовочный файл **square.h** может быть включен в нашу программу с помощью директивы `#include "square.h"`, размещенной в верхней части программы. Директива **#include** создаёт копию указанного файла, которая включается в программу вместо директивы. Существует две формы использования директивы **#include**:

#inclue <filename>
#include "filename"

Разница между ними заключается в том, где препроцессор будет искать файлы, которые необходимо включить. Если имя заключено в кавычки, препроцессор ищет его в том же каталоге, что и компилируемый файл. Такую запись обычно используют для включения определённых пользователем заголовочных файлов. Если же имя файла заключено в угловые скобки – используемые для *файлов стандартной библиотеки,* то поиск будет вестись в зависимости от конкретной реализации компилятора, обычно в предопределённых каталогах.

Директива **`#include`** чаще всего используется для включения заголовочных файлов стандартной библиотеки, таких как **`stdio.h, stdlib.h`**. Директива **`#include`** так же используется с программами, состоящими из нескольких файлов, которые необходимо компилировать вместе. Часто с помощью этой директивы включается заголовочный файл, содержащий общие для отдельных файлов программы объявления.

Стандартные библиотеки C++

1. Ошибки <errno.h>
2. Стандартные определения <stddef.h>
3. Диагностика <assert.h>
4. Обработка символов <ctype.h>
5. Локализация <locale.h>
6. Математика <math.h>
7. Нелокальные переходы <setjmp.h>
8. Обработка сигналов <signal.h>
9. Переменные списки аргументов <stdarg.h>
10. Ввод/вывод <stdio.h>
11. Утилиты общего назначения <stdlib.h>
12. Обработка строк <string.h>
13. Дата и время <time.h>
14. Ограничения реализации <limits.h>

Список литературы

1. Конспект лекций по ООП (Торхова Н.А.);
2. Франк. Учебный курс С++;
3. Х.М. Дейтел, Дж. Дейтел “Как программировать на С++”;
4. Керниган, Д. Ричи “Язык программирования С”;
5. Бочков, Субботин “Язык программирования С++”;

Содержание