**лекция 9**

**Тема лекции:** Управление памятью. Почленное присвоения. Шаблонные функции и шаблонные классы

**Перегрузка и управления памятью**

Операторы new и delete могут быть перегруженными с целью нанесения новых возможностей управления памятью для объектов класса. Перегрузка new в объявлении класса указывает компилятору, что отныне ответственность за выделение памяти для объектов класса лежит на программисте. Для перегрузки new надо использовать прототип функции вида

void \* operator new (size\_t size)

В дальнейшем обращение к оператору new для выделения памяти объектам класса будут перенаправлены в перегруженной функции. Функция должна возвращать адрес области памяти, выделенной объекта. Если необходимого пространства нет, функция должна возвращать null.

Пример: Вместо кучи, программа выделяет память объектам в глобальном буфере.

class BrandNew

{

private:

int x;

public:

BrandNew ();

void \* operator new (size\_t size)

};

char buf [512];

int index = 1;

main ()

{

cout << "\ nСтворений локальный объект";

BrandNew b1;

cout << "\ n Расположение пространства, выделенный new";

BrandNew \* b2 = new BrandNew;

BrandNew \* b3 = new BrandNew;

BrandNew \* b4 = new BrandNew;

BrandNew \* b5 = new BrandNew;

return 0;

}

BrandNew :: BrandNew ()

{

x = index;

}

void \* BrandNew :: operator new (size\_t size)

{

if (index> = 512-sizeof (BrandNew)) return 0;

else

{

int k = index;

index + = sizeof (BrandNew)

return & buf [k];

}

}

Здесь функция перегрузки оператора new проверяет наличие пространства в глобальном буфере. Если его нет, функция возвращает 0, что является основанием для возвращения оператором new значение NULL. Если свободное пространство есть, то глобальный индекс увеличивается на размер необходимой памяти, передается функции в параметре size. Затем функция возвращает адрес выделенного участка памяти. Можно перегрузить delete для отслеживания удаления объектов, которые адресованы указателями. Прототип функции перегрузки оператора delete должен иметь вид:

void operator delete (void \* p)

где p ссылается на объект удаляется. Можно сказать функцию еще и так:

void operator delete (void \* p, size\_t size)

В этом случае С ++ будет дополнительно передавать функции число байтов в удаленном объекте. В рассматриваемой программе для добавления функции перегрузки оператора delete к классу BrandNew нужно объявить функцию-член в открытой части класса:

void operator delete (void \* p)

{

cout << "\ n Объект, удаляется -" << г;

}

Оператор вывода будет отображать адрес каждого объекта удаляется. Перегружен оператор delete на самом деле не освобождает никакой памяти, потому что объекты не запоминаются в купе. Для удаления нескольких объектов потребуется несколько операторов delete:

delete b2;

delete b3;

delete b4;

delete b5; // (Перед оператором return в main ())

Чтобы использовать средства управления памятью С ++ для выделения пространства в куче для объектов, в которых перегружен оператор new, нужно поставить перед оператором двойную двоеточие. Например, в строке

BrandNew \* x = :: new BrandNew;

игнорируется перегружен в объектах класса BrandNew оператор new. Аналогично, :: delete служит для обращения к оператору delete, используемый по умолчанию.

Обычно, если new не может выполнить запрос на выделение памяти, оператор возвращает NULL. Для того, чтобы изменить действия, выполняемые по умолчанию, надо присвоить адрес функции-обработчика ошибок указателе \_new\_handler, определенном таким образом:

typedef void (\* vfp) (void);

vfp \_new\_handler;

Использование typedef не обязательно, но оно облегчает чтение объявления. Функция-обработчик ошибок ничего не возвращает и не имеет аргументов. Она устанавливается с помощью вызова функции set\_new\_handler (). Ее прототип содержится в файле new.h:

vfp set\_new\_handler (vfp)

Обработчик ошибок реализуется так же, как и любая другая функция С ++. Например, можно аварийно завершить программу с сообщением об ошибке нехватки памяти:

void memerr (void)

{

fputs ( "\ n \ nOut of memory \ n", stderr)

exit (1);

}

Чтобы эта функция вызывалась в случае возникновения ошибок, связанных с нехваткой памяти, нужно передать ее адрес функции set\_new\_handler ():

set\_new\_handler (memerr)

**Почленное присвоение**

Присвоение объекта класса другому объекту может привести к тому, что члены-указатели на данные будут ссылаться на тот же участок памяти. Рассмотрим фрагмент программы:

TAnyClass V1;

TAnyClass V2 (1, 2, "Строка");

V1 = V2;

После инициализации объекта V1 значениями по умолчанию и определение V2 с явными аргументами, оператор присваивания копирует V2 в V1. Но присвоение не вызывает ни конструктора класса, даже конструктора копии. Если в классе объявляются или наследуются члены-указатели на данные, скопированные указатели будут ссылаться на тот же участок памяти.

Конструктор копии не решит этой проблемы, поскольку задействованы объекты уже созданы. Решение проблемы - перегрузка оператора присваивания "=". Надо использовать объявления вида void operator = (const class &). К примеру:

void operator = (const TAnyClass & copy)

Здесь параметр ссылается на объект копируется. Реализация этой функции:

void TAnyClass :: operator = (const TAnyClass & copy)

{

cout << "....

if (this = & copy) return;

delete s;

i = copy.i;

r = copy.r;

s = strdup (copy. s)

}

В операторе if сравнивается скрытый указатель this и адрес параметра copy, на который он ссылается. Это необходимо для предотвращения попытки присвоения объекта самому себе. Во второй части оператора if освобождается участок памяти, адресуется указателем s. Дальнейшее код перегруженной функции аналогичный кода конструктора копии. Конструктор копии можно заставить работать совместно с перегруженным оператором "=". Для этого внутри конструктора копии надо вызвать функцию перегрузки operator = присвоением объекта копируется, объекту this.

TAnyClass :: TAnyClass (TAnyClass & copy)

{

s = null;

\* This = copy;

}

Очень важно инициализировать все члены, проверяет перегружен оператор присвоения. Например, если указатель содержит мусора вместо NULL, функция перегрузки оператора пытается освободить память, которая не была выделена.

**шаблоны**

Подобно тому, как класс является схематичным описанием построения объекта, так и шаблон является схематичным описанием построения функций и классов. Шаблоны указывают только спецификации функций и классов, но не детали действительной реализации.

**шаблонные функции**

Шаблоны функций обычно объявляют в заголовочный файл и они имеют такой общий вид:

template <class T> void f (Т param)

{

// Тело функции

}

Шаблонная функция начинается строкой template <class T>, который указывает компилятору, что Т - обусловлен пользователем тип функции. Необходим хотя бы один параметр типа T для передачи функции данных для обработки. Можно задать указатель (Т \* param) или посылку (T & param). Функция может объявлять несколько параметвров и возвращать значение типа T:

template <class T> T f (int a, T b)

{

...

}

В этой версии шаблонная функция f () возвращает значение типа T и имеет два параметра - целое а и неопределенный объект b.

Пользователи шаблона укажут действительный тип данных для T. Например, в программе можно задать такой прототип:

double f (int a, double b)

Необходимо обеспечить реализацию этой функции, если она - обычная. Но, поскольку f () - шаблонная функция, компилятор реализует код функции, заменив T на double.

пример:

template <class T> T max (T a, T b)

{

if (a> b) return a;

else

return b;

}

Здесь T имеет неопределенный тип, объект которого возвращает шаблонная функция max (). Функции max () необходимы два аргумента типа T. Операторы функции - это схема для реальных операторов, которые будут сгенерированы позже, когда будет задан действительный тип T. Можно использовать более одного неопределенный тип:

template <class T1, class T2> T1 max (T1 a, T2 b)

В этой версии функция max () будет возвращать значение типа T1 и ей необхиднi два аргумента: один типа T1, второй - типа T2. В программе, которая использует шаблонные функции, необходимо отметить их прототипы, которые компилятор использует для создания настоящих тел функций.

**шаблонные классы**

Шаблон класса обеспечивает скелет обобщенного класса для его последующей реализации. Чаще всего шаблоны классов объявляют в заголовочный файл. Рассмотрим пример создания шаблонного класса. Создадим заголовочный файл db.h:

template <class T> class TDataBase

{

private:

T \* rp;

int num;

public:

TDataBase (int n)

{

rp = new T [num = n];

}

~ TDataBase ()

{

delete [] rp;

}

T & GetRecord (int recnum)

};

template <class T>

T & ТDataBase <T> :: GetRecord (int recnum)

{

T \* crp = rp;

if (0 <= recnum && recnum <num)

while (recnum--> 0)

crp ++;

return \* crp;

}

Здесь объявления шаблона класса выглядит так:

template <class T> class TDataBase

{

 ...

};

где T - неопределенный тип, задаваемый пользователем шаблона. Вместо T можно использовать любой другой идентификатор. T можно впоследствии заменить любым встроенным типом, другим классом, указателем и т.д. TDataBase - имя шаблонного класса. Для ясности заголовок шаблона лучше объявлять в отдельных строках:

template <class T>

class AnyClass

{

 ...

};

Можно также отметить несколько типов:

template <class T1, class T2, class T3>

class TAnotherClass

{

 ...

};

В шаблоне класса «class T> тип Т можно использовать для объявления данных-членов типов возвращаемых значений функций членами, параметров и других элементов неопределенных типов. Например, в объявлении класса TDataBase объявляется указатель типа T с именем rp:

T \* rp;

На этой стадии истинная природа T еще неизвестна, поэтому в программе его можно использовать только в общих случаях. Однако, конструктор TDataBase () выделяет память для массива объектов T, присваивая адрес массива указателе rp и заодно устанавливая член num равным необходимому числу записей. В деструкторе этот массив удалится.

Функция-член GetRecord () возвращает посылку на объект типа T, идентифицируется номером записи recnum. Это еще одна обобщенная операция, не требующая задачи определенного типа T.

Функции-члены шаблонного класса могут быть встроенными или реализоваться отдельно. И поскольку эти функции-члены шаблонного класса - только объявление, то их можно поместить в заголовочный файл. Заголовок функции члена предваряется фразой template <class T>, затем идет тип значения, которое возвращается, имя класса и оператор области видения. В последние объявляется сама функция и ее тело.

Рассмотрим функцию GetRecord (). Она возвращает посылку на объект типа T. Внутри функции указатель crp типа T присваивается указателю того же типа с именем rp. То есть, указатель crp ссылается на первую запись, сохраненный в объекте класса TDataBase. Далее проверяется, соответствует ли параметр recnum допустимому диапазону значений. Если так, то цикл while декрементуе параметр recnum до 0 и одновременно передвигает указатель crp на одну запись в базе данных.

**В классе ТDataBase допустимый выражение**

crp ++;

даже несмотря на то, что тип объекта, на который ссылается crp, неизвестно. Позже, когда будет задан действительный тип для шаблона класса, компилятор сможет сгенерировать соответствующие инструкции для увеличения указателя crp на размер sizeof (T).

Далее в программе возвращается розименоване значение указателя crp, то есть посылка на любой объект, адресуется crp. Это завершает объявление шаблона класса, в котором не делается никаких предположений о том, какого типа данные в нем запоминаются.

Теперь используем этот шаблон для создания объекта класса базы данных, способного запомнить некоторое число записей.

class TRecord

{

private:

char name [41];

public:

TRecord (const char \* s)

{

Assign (s)

}

void Assign (const char \* s)

{

strncpy (name, s, 40);

}

char \* getName (void)

{

return name;

}

};

main ()

{

int rn;

TDataBase <TRecord> db (3);

TDataBase <Trecord \*> dbp (3);

TDataBase <TRecord> \* pdb;

TDataBase <Trecord \*> \* ppdb;

cout << "\ n \ nDatabase of 3 TRecords \ n";

db.GetRecord (0) .Assign ( "One")

db.GetRecord (1) .Assign ( "Two")

db.GetRecord (2) .Assign ( "Three")

for (rn = 0 rn <= 2; rn ++)

cout << db.GetRecord (rn) .GetName () << '\ n';

cout << "\ n \ nDatabase of 3 TRecord pointers \ n";

dbp.GetRecord (0) = new TRecord ( "One string")

dbp.GetRecord (1) = new TRecord ( "Two string")

dbp.GetRecord (2) = new TRecord ( "Three string")

for (rn = 0 rn <= 2; rn ++)

cout << dbp.GetRecord (rn) -> GetName () << '\ n';

cout << "\ n \ nPointer to database of 3 TRecords \ n";

pdb = new TDataBase <TRecord> (3);

pdb-> GetRecord (0) .Assign ( "string one")

pdb-> GetRecord (1) .Assign ( "string two")

pdb-> GetRecord (2) .Assign ( "string three")

for (rn = 0 rn <= 2; rn ++)

cout << pdb-> GetRecord (rn) .GetName () << '\ n';

ppdb = new TDataBase <TRecord \*> (3);

ppdb-> GetRecord (0) = new TRecord ( "string1")

ppdb-> GetRecord (1) = new TRecord ( "string2")

ppdb-> GetRecord (2) = new TRecord ( "string3")

for (rn = 0 rn <= 2; rn ++)

cout << ppdb-> GetRecord (rn) -> GetName () << '/ n';

getch ();

return 0;

}

В программе демонстрируется 4 средства создания объекта класса с помощью шаблона класса. В строке

TDataBase <TRecord> db (3);

определяется объект с именем db шаблонного класса типа TDataBase и задается TRecord в качестве класса, замещающего T в шаблоне. Выражение в скобках (3) - инициализатор db, передан конструктору класса TDataBase. Для создания базы данных необязательно использовать классы. Например, создадим базу данных из 100 действительных значений двойной точности:

TDataBase <double> dbd (100)

Поскольку класс TDataBase написан для хранения объектов произвольных типов, он называется контейнерным классом. Лучшие контейнерные классы вполне обобщенными.

Далее в программе создаются другие экземпляры шаблонного класса:

• объект dbp класса TDataBase, состоящий из 3х указателей на Trecord;

• указатель pdb на объект класса TDataBase с незаданные числом объектов типа TRecord;

• указатель ppdb на базу данных указателей на TRecord.

В программе объект класса TDataBase используется так же, как и любой другой не шаблонный объект.

Во всех приведенных примерах не надо информировать компилятор о типах данных, использованных классом TDataBase, с помощью приведения типов.

Рекомендуется не писать шаблоны с чистого листа. Вместо этого пишется класс, использует заданные объекты. Затем, после его наладка, можно превратить его в универсальный шаблон. Это помогает определить истинно обобщенные свойства классов.