Объектно-ориентированное программирование с использованием языка C++

компьютерная академия ШАГ

Урок №6

Объектно-ориентированное программирование с использованием языка C++

Содержание

Перегрузка оператора ->

Мы надеемся, что вы помните, что в C++ можно перегрузить почти все операторы, за исключением нескольких. Во всяком случае, оператор -> перегружается, и это имеет значение крайне важное. Кстати, этот оператор называется селектором (member selector). Рассмотрим пример:

```
#include <iostream>
using namespace std;
//класс, указатель на который
//будет инкапсулирован
class Temp
{
    int TEMP;
    public:
        //конструктор
        Temp(){TEMP=25;}
        //функция показа на экран
        void TempFunction(){
            cout<<"TEMP = "<<TEMP<<"\n\n";
        }
        //функция установки значения
        void TempSet(int T){
            TEMP=T;
        }
};

//класс, инкапсулирующий указатель
class MyPtr
{
    //указатель на класс
    Temp Temp*ptr;
```

```
    public:
        //конструктор
        MyPtr(Temp*p=NULL){
            ptr=p;
        }

        //Оператор преобразования типа
        //от инкапсулированного к инкапсулирующему
        operator Temp*(){
            return ptr;
        };
        //Оператор селектора ->
        //который позволит обратиться
        //напрямую к «спрятанному»
        //указателю
        Temp* operator->(){
            return ptr;
        }
        //оператор ++ для смещения указателя вперед
        MyPtr operator++(){
            ptr++;
            return *this;
        }
};

void main ()
{
    //создание нового объекта
    Temp*main_ptr = new Temp;
    //простое обращение к членам
    //объекта через «родной» указатель
    main_ptr->TempFunction();

    //создание объекта класса-указателя
    MyPtr pTemp(main_ptr);
    //обращение через класс-указатель
    pTemp->TempFunction();
```

```
    //создание массива объектов
    //инкапсулируемого класса
    Temp*arr_=new Temp[3];

    //заполнение вышеозначенного массива
    //значениями от 0 до 2
    for(int i=0;i<3;i++) arr_[i].TempSet(i);

    //создание объекта класса указателя
    //и запись в него адреса массива
    //(здесь работает преобразование типа)
    MyPtr arr_temp=arr_;
    //сдвиг на один элемент вперед
    arr_temp++;
    //демонстрация результата
    arr_temp->TempFunction();

    //удаление объектов
    delete main_ptr;
    delete[]arr_;
}

Результат работы программы
TEMP = 25
TEMP = 25
TEMP = 1
```

Итак, обсудим результат: У нас есть класс Temp, который может иметь экземпляры, обладает некоторым членом TEMP и минимальным набором функций для работы с ним. Кроме того, мы создали класс объекта-указателя

MyPtr, в котором храним обычный указатель, но доступ к нему ограничиваем, и перегружаем для него операторы:

1. Оператор приведения типа от Temp к MyPtr
2. Оператор -> (selector).
3. Оператор ++, реализующий сдвиг указателя на один шаг вперед.

Рассмотрим положительные моменты — мы получили класс объектов-указателей, которые можно смело применять вместо настоящих. Это удобно, т.к. все функции для работы с указателем можно инкапсулировать в этом классе. Однако есть еще одно широко распространенное применение данной конструкции, с которым мы с вами познакомимся в следующем разделе урока. Вперед!

Понятие «умного» указателя (smart pointer)

То о чём мы с Вами сейчас будем говорить, реализуется с помощью перегрузки селектора. Итак.

Умный указатель (англ. smart pointer) — класс, имитирующий интерфейс обычного указателя и добавляющий к нему некую новую функциональность, например проверку границ при доступе или очистку памяти.

Иначе говоря, умные указатели — это объекты, в которых хранится указатель на настоящий объект и, как правило, счётчик числа обращений к объекту.

В классе также присутствует деструктор для настоящего объекта, который не вызывается извне, а только внутри smart pointer. Принцип таков, что, когда счётчик обращений к объекту равен 0, вызовется деструктор.

Например, есть массив указателей на объекты. Вы используете копию указателя на один объект где-нибудь ещё. Потом очищаете массив, удаляя при этом объекты, а "где-нибудь ещё" все ещё хранит указатель. Естественно, что, при доступе к нему происходит сбой программы, т.к. объект был удалён при очистке массива.

При наличии smart pointer, такого не произойдёт, т.к. при создании копии указателя внутренний счётчик увеличится на 1. Те будет равен двум — 1 для массива, и ещё 1 для копии. Теперь даже если мы удалим объект из массива, внутренний счётчик уменьшится на 1 и ста-

нет равным 1, т.е. копия может продолжать работать с объектом. После того как он станет ненужным, копия освобождает память от него и счётчик становится равным 0. Именно, в этот момент вызывается деструктор для оригинального объекта.

Ну что же, всё сказано. Пора попробовать блюдо под названием «умный указатель» на вкус. :)))

```
#include <iostream>
using namespace std;
class Temp
{
     int TEMP;
     public:
          //конструктор
          Temp(){TEMP=25;}
          //функция показа на экран
          void TempFunction(){
               cout<<"TEMP = "<<TEMP<<"\n\n";
          }
          //функция установки значения
          void TempSet(int T){
               TEMP=T;
          }
};

//Класс, реализующий умный указатель
class SmartPointer
{
     //Инкапсулированный указатель
     Temp*ptr;
     //счётчик копий
     int count_copy;

     public:
          //конструктор
```

```
        SmartPointer (Temp*p=NULL){
            //записываем 0 при создании копий нет
            count_copy=0;
            ptr=p;
        }
        //конструктор копирования
        SmartPointer (const SmartPointer&obj){
            //создается копия - увеличиваем счётчик
            ptr=obj.ptr;
            count_copy++;
        }
        //перегрузка оператора равно
        SmartPointer operator=(const SmartPointer&obj){
            //создается копия - увеличиваем счётчик
            ptr=obj.ptr;
            ptr=obj.ptr;
            count_copy++;
            //возвращаем сам объект для ситуации a=b=c
            return *this;
        }
        //уничтожение объекта
        ~SmartPointer(){
            //если объект есть и копий нет
            if(ptr!=NULL&&count_copy==0){
                cout<<"\n~Delete Object\n";
                //уничтожаем объект
                delete[]ptr;
            }
            //в противном случае(уничтожается копия)
            else{
                //уменьшаем счётчик
                count_copy--;
                cout<<"\n~Delete Copy\n";
            }
        }
        //старая добрая перегрузка селектора
        Temp* operator->(){
            return ptr;
        }
};
```

```
void main(){
    //создаем объект
    Temp*main_ptr=new Temp;
    //инициализируем этим объектом умный указатель
    SmartPointer PTR(main_ptr);
    //проверяем работу умного указателя
    PTR->TempSet(100);
    PTR->TempFunction();
    //создаем копию (работа конструктора копирования)
    SmartPointer PTR2=PTR;
}

Результат работы программы:
//работа с объектом через умный указатель
TEMP = 100
//уничтожение копии
~Delete Copy
//уничтожение самого объекта
~Delete Object
```

Сейчас, мы создали свой собственный указатель, в последующих уроках, нами будет рассмотрен уже готовый smart poiter из библиотеки STL, под названием auto_ptr.

Функции с произвольным количеством и типом аргументов

Данный раздел урока посвящен изучению относительно нового типа функций, количество параметров у которых становится известным только в момент обращения к этой функции.

Итак, в языке C++ допустимы функции, у которых количество параметров и их типы при компиляции самой функции не определены. Эти значения становятся известными только в момент вызова функции, когда явно задан список фактических параметров. При определении и описании таких функций, имеющих списки параметров неизвестной длины, спецификация формальных параметров заканчивается многоточием. То есть, общий синтаксис таков:

```
тип_функции имя_функции (спецификация_явных_ параметров,...);
```

Здесь спецификация явных параметров, количество и типы которых фиксированы и известны в момент компиляции. Эти параметры называются обязательными. После списка явных (обязательных) параметров ставится необязательная запятая, а затем многоточие, извещающее компилятор, что дальнейший контроль соответствия количества и типов параметров при обработке вызова функции проводить не нужно.

Каждая функция с переменным списком параметров должна иметь механизм определения их количества и типов. Существует два подхода к решению этой задачи.

1. Первый подход предполагает добавление в конец списка необязательных параметров специального параметра-индикатора с уникальным значением, которое будет сигнализировать об окончании списка. В теле функции параметры последовательно перебираются, и их значения сравниваются с заранее известным концевым признаком.
2. Второй подход предусматривает передачу в функцию значения реального количества фактических параметров. Эту величину можно передавать в функцию с помощью одного из явно задаваемых (обязательных) параметров.

Следует отметить, что, в обоих подходах — и при задании концевого признака, и при указании числа реально используемых фактических параметров — переход от одного фактического параметра к другому выполняется с помощью указателей, то есть с использованием адресной арифметики. Проиллюстрируем сказанное примерами.

Пример 1. *Составить программу вычисления суммы заданных целых чисел с использованием количества слагаемых.*

```
#include <iostream>
using namespace std;

//Прототип функции.
long summa (int,...);
```

```
void main()
{
    cout << "\n summa(2,4,6)=" << summa (2,4,6);

    cout << "\n summa(6,1,2,3,4,5,6)="
                                  << summa (6,1,2,3,4,5,6);
}
//Передаем количество параметров.
long summa (int k,...)
{
    //pk содержит адрес расположения
    //начала списка параметров.
    //это связано с тем, что параметры
    //располагаются по порядку
    //в оперативной памяти
    int *pk=&k;

    //подсчет суммы
    //к - количество параметров
    long sm=0;
    for (;k;k--)
        sm+=*(++pk);

    return sm;
}
```

Комментарии к программе

В обычном случае при описании функции задается список переменных — набор формальных параметров, используя которые осуществляется доступ к переданным значениям. Здесь такого списка нет. Каким же образом получить доступ к перечню переданных параметров? Это осуществляется с помощью указателей: сначала в указатель помещается адрес конца или начала списка явных

параметров, а затем, используя это значение, мы производим перемещение по переменному списку параметров.

Пример 2. *Перепишем программу, используя предопределенное значение (пусть это будет 0).*

```
#include <iostream>
using namespace std;
//k - это теперь, всего лишь один
//из суммируемых параметров
long summa (int k,...)
{
    //начиная с k
    int *pk=&k;
    long sm=0;
    //движемся до тех пор,
    //пока не встретим значение 0
    while (*pk)
        //подсчет суммы
        sm+=*(pk++);
    return sm;
}
void main()
{
    //тестируем
    cout << "\n summa(4,6,0)="<< summa (4,6,0);
    cout << "\n summa(1,2,3,4,5,6,0)="
                                << summa (1,2,3,4,5,6,0);
    cout << "\n summa(1,2,0,4,5,6,0)="
                                << summa (1,2,0,4,5,6,0);
}

Результат работы программы:

summa(4,6,0)=10
summa(1,2,3,4,5,6,0)=21

//Внимание здесь функция
```

```
//доработает лишь до первого нуля
//остальные данные будут утеряны.
summa(1,2,0,4,5,6,0)=3
```

Есть ещё один способ проанализировать аргументы внутри функции. Именно этот способ позволит анализировать аргументы не только произвольного количества, но и с разными типами. Для начала подключаем библиотеку stdarg.h, именно в ней находятся, необходимые автоматические средства анализа. Разберём последующие действия по шагам:

1. Объявляем переменную типа va_list (переопределённый char*). Именно здесь мы будем хранить указатель на список параметров нашей функции.
2. Единообразие вызываем функцию:

```
void va_start( va_list arg_ptr, prev_param );
```

Первый параметр — созданная заранее переменная типа va_list, в неё запишется указатель на список параметров, созданный данной функцией.
Второй параметр нашей функции, что бы va_list могла от чего-то оттолкнуться при связывании arg_ptr и списка.

3. Анализируем параметры с помощью многократного вызова функции:

```
type va_arg( va_list arg_ptr, type );
```

Данная функция получает параметр из списка, указатель на который передан в неё в качестве первого

параметра, и переставляет этот указатель на следующий элемент. В качестве второго параметра, который необходимо достать из списка..

4. Заканчиваем анализ параметров и очищаем внутренний указатель, устанавливая его в нуль с помощью функции:

```
void va_end( va_list arg_ptr );
```

Рассмотрим пример:

```
#include <iostream>
using namespace std;
#include <stdarg.h>
//функция подсчитывает сумму чисел количеством count
//typeof определяет, какого типа будут элементы
//true - параметры целого типа (int)
//false - параметры целого типа (double)
void AnyType(int count,bool typeof,...){
    //сумма для целочисленных параметров
    int sumi=0;
    //сумма для вещественных параметров
    double sumd=0.0;
    //создаём указатель на список параметров
    va_list arg_ptr;
    //получаем этот указатель, отталкиваясь
    //от первого фактического параметра
    va_start(arg_ptr,count);
    //пропускаем второй фактический параметр
    va_arg(arg_ptr,bool);
    //count раз считываем последующие параметры
    while(count--){
        //если typeof - true, то считываем параметры
        //типа int
        //если typeof - false, то считываем параметры
        //типа double
```

```
        (typeof)?sumi+=va_arg(arg_ptr,int):
                        sumd+=va_arg(arg_ptr,double);
    }
    //закрываем список параметров
    va_end(arg_ptr);

    //если typeof - true, то показываем сумму типа int
    //если typeof - false, то показываем сумму
    //типа double
    (typeof)?cout<<"Integer sum =
                "<<sumi:cout<<"Double sum = "<<sumd;
    cout<<"\n\n";
}
void main()
{
    //вызываем функцию для суммирования
    //параметров типа int
    AnyType(4,true,3,8,9,4);
    //вызываем функцию для суммирования
    //параметров типа double
    AnyType(3,false,2.5,1.1,3.6);
}
```

Как вы уже успели заметить, функции с неограниченным количеством параметров имеют ряд недостатков. В частности, анализ параметров, является затруднённым и часто даже имеет огромный код. Кроме того, вероятность ошибки на этапе выполнения для таких функций повышается во много раз, так как за правильной передачей параметров очень трудно уследить. Поэтому, мы рекомендуем Вам относиться к реализации этого аппарата крайне внимательно.

Домашнее задание

1. Написать класс, реализующий работу с комплексными числами.
2. Написать программу, которая на основе классов реализует карточную игру Блэк-Джек.
3. Написать программу, которая реализует функцию с неограниченным количеством параметров, организовывающую форматированный вывод на экран.