Глава 13

Указатели на объекты

В этой главе...

- Массивы объектов
- Указатели на объекты
- Строгая типизация
- Списки объектов

Программисты на C++ постоянно создают массивы чего-либо. Формируются массивы целочисленных значений, массивы действительных значений; так почему бы не создать массив студентов? Студенты все время находятся в списках (причем гораздо чаще, чем им хотелось бы). Концепция объектов `Student`, стройными рядами ожидающих своей очереди, слишком привлекательна, чтобы можно было пройти мимо нее.

Объявление массивов объектов

Массивы объектов работают так же, как и массивы простых переменных. (В главе 7, "Хранение последовательностей в массивах", описаны основы работы с массивами простых (встроенных) типов, а в главах 8, "Первое знакомство с указателями в C++", и 9, "Второе знакомство с указателями в C++", подробно рассматриваются указатели.) В качестве примера можно использовать следующий фрагмент программы `ArrayOfStudents`:

```
// ArrayOfStudents - определение массива объектов
//                   Student и обращение к его
//                   элементам

#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
public:
    int    semesterHours;
    double gpa;
    double addCourse(int hours, double grade) { return 0.0; };
};

void someFn()
{
    // Объявляем массив из 10 студентов
    Students[10];
```

```
    // Пятый студент получает 5.0 (повезло!)
    s[4].gpa = 5.0;
    s[4].semesterHours = 32;

    // Добавим еще один курс пятому студенту,
    // который на этот раз провалился...
    s[4].addCourse(3, 0.0);
}
```

В данном фрагменте `s` является массивом объектов типа `Student`. Запись `s[4]` означает пятый элемент массива, а значит, `s[4].gpa` является усредненной оценкой пятого студента. В следующей строке с помощью функции `s[4].addCourse()` пятому студенту добавляется еще один прослушанный и несданный курс.

Объявление указателей на объекты

Указатели на объекты работают так же, как и указатели на простые типы:

```
// ObjPtr - определение и использование
//          указателя на объект Student
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  public:
    int    semesterHours;
    double gpa;
    double addCourse(int hours, double grade);
};

int main(int nNumberofArgs, char* pszArgs[])
{
    // Создание объекта Student
    Student s;
    s.gpa = 3.0;

    // Создание указателя на объект Student
    Student* pS;

    // Заставляем указатель указывать на наш объект
    pS = &s;
    cout << "s.gpa   = " << s.gpa   << "\n"
         << "pS->gpa = " << pS->gpa << endl;

    // Ожидание, позволяющее пользователю увидеть результат
    // работы программы
    cout << "Нажмите Enter для продолжения..." << endl;
    cin.ignore(10, '\n');
    cin.get();
    return 0;
}
```

В программе объявляется переменная s типа Student, после чего создается переменная pS, которая является "указателем на объект типа Student"; другими словами, указателем Student*. Программа инициализирует значение одного из членов-данных s и присваивает адрес s переменной pS. Затем программа обращается к объекту s: один раз — по имени, а затем — с использованием указателя на объект. Странную запись pS->gpa я объясню немного позже в этой главе.

Разыменование указателей на объекты

По аналогии с указателями на простые переменные можно решить, что в приведенном ниже примере происходит обращение к усредненной оценке студента s:

```
int main(int nNumberofArgs, char* pszArgs[])
{
    // Этот пример некорректен
    Student s;
    Student* pS= &s; // Создаем указатель на объект s

    // Обращаемся к члену gpa объекта, на который
    // указывает pS (этот фрагмент неверен)
    *pS.gpa = 3.5;

    return 0;
}
```

Как сказано в комментарии, этот код работать не будет. Проблема в том, что оператор . будет выполнен раньше оператора *. Таким образом, *pS.gpa интерпретируется как *(pS.gpa). Скобки необходимы для того, чтобы заставить оператор разыменования вычисляться до оператора точки:

```
int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    Student* pS= &s; // Создаем указатель на объект s

    // Обращаемся к члену gpa того объекта, на который
    // указывает pS (теперь все работает правильно)
    (*pS).gpa = 3.5;

    return 0;
}
```

Теперь *pS вычисляет объект, на который указывает pS, а следовательно, .gpa обращается к члену этого объекта.

Использование стрелок

Использование для разыменования указателей на объекты оператора * со скобками будет прекрасно работать. Однако даже самые твердолобые программисты скажут вам, что такой синтаксис разыменования очень неудобен.

Для доступа к членам объекта C++ предоставляет более удобный оператор ->, позволяющий избежать неуклюжей конструкции со скобками и оператором *; таким образом, pS->gpa эквивалентно (*pS).gpa. В результате получаем следующий преобразованный код рассмотренной программы:

```
int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    Student* pS= &s; // Создаем указатель на объект s

    // Обращаемся к члену gpa того объекта, на который
    // указывает pS (теперь все работает правильно)
    pS->gpa = 3.5;

    return 0;
}
```

Этот оператор используется гораздо чаще, поскольку его легче читать (хотя обе формы записи совершенно тождественны).

Передача объектов функциям

Передача указателей функциям — один из множества способов выразить себя в области указателей.

Вызов функции с передачей объекта по значению

Как вы знаете, C++ передает аргументы в функцию по ссылке при использовании в описании символа `&` (см. главу 8, "Первое знакомство с указателями в C++"). Однако по умолчанию C++ передает функции только значения аргументов. (Обратитесь к главе 6, "Создание функций", если вы этого не знали.)

Объекты сложных пользовательских классов передаются точно так же, как и простые `int`, как видно из приведенной ниже программы `PassObjVal`:

```
// PassObjVal - попытка изменить значение объекта в функции
//              оказывается неудачной при передаче объекта
//              по значению
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  public:
    int    semesterHours;
    double gpa;
};

void someFn(Student copyS)
{
    copyS.semesterHours = 10;
    copyS.gpa           = 3.0;
    cout << "Значение copyS.gpa = "
         << copyS.gpa << "\n";
}

int main(int nNumberofArgs, char* pszArgs[])
{
```

```
    Student s;
    s.gpa = 0.0;

    // Вывод значения s.gpa до вызова someFn()
    cout << "Значение s.gpa = " << s.gpa << "\n";

    // Передача существующего объекта
    cout << "Вызов someFn(Student)\n";
    someFn(s);
    cout << "Возврат из someFn(Student)\n";

    // Значение s.gpa остается равным 0
    cout << "Значение s.gpa = " << s.gpa << "\n";

    // Ожидание, позволяющее пользователю увидеть результат
    // работы программы
    cout << "Нажмите Enter для продолжения..." << endl;
    cin.ignore(10, '\n');
    cin.get();
    return 0;
}
```

В этом примере функция `main()` создает объект `s`, а затем передает его в функцию `someFn()`.

Осуществляется передача по значению не самого объекта, а его копии.

Объект `copyS` начинает свое существование внутри функции `someFn()` и является точной копией объекта `s` из `main()`. При этом любые изменения содержимого объекта `copyS` никак не отражаются на объекте `s` из функции `main()`. Вот что дает программа на выходе:

```
Значение s.gpa = 0
Вызов someFn(Student)
Значение copyS.gpa = 3
Возврат из someFn(Student)
Значение s.gpa = 0
Нажмите Enter для продолжения...
```

Вызов функции с передачей указателя на объект

Зачастую программисту требуется, чтобы изменения переданного объекта в функции отражались и в вызывающей функции. В таком случае вместо того, чтобы передавать объект по значению, можно передавать в функцию адрес объекта или ссылку на него:

```
// PassObjPtr - изменение значения объекта в функции
//              при передаче указателя на объект
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;
```

```
class Student
{
  public:
    int    semesterHours;
    double gpa;
};

void someFn(Student* pS)
{
    pS->semesterHours = 10;
    pS->gpa           = 3.0;
    cout << "Значение pS->gpa = "
         << pS->gpa << "\n";
}

int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    s.gpa = 0.0;

    // Вывод значения s.gpa до вызова someFn()
    cout << "Значение s.gpa = " << s.gpa << "\n";

    // Передача существующего объекта
    cout << "Вызов someFn(Student*)\n";
    someFn(&s);
    cout << "Возврат из someFn(Student*)\n";

    // Значение s.gpa теперь равно 3.0
    cout << "Значение s.gpa = " << s.gpa << "\n";

    // Ожидание, позволяющее пользователю увидеть результат
    // работы программы
    cout << "Нажмите Enter для продолжения..." << endl;
    cin.ignore(10, '\n');
    cin.get();
    return 0;
}
```

В этом примере аргумент, передаваемый в `someFn()`, имеет тип указателя на объект `Student`, что записывается как `Student*` (это отражает способ вызова программой функции `someFn()` с передачей адреса `s` вместо самого объекта). Теперь вместо значения объекта `s` в функцию `someFn()` передается указатель на объект `s`, что позволяет изменять состояние этого объекта. Соответственно изменяется и способ обращения к аргументам функции внутри ее тела: теперь для разыменования указателя `pS` используются операторы-стрелки.

На этот раз вывод программы имеет следующий вид:

```
Значение s.gpa = 0
Вызов someFn(Student*)
Значение pS->gpa = 3
Возврат из someFn(Student*)
Значение s.gpa = 3
Нажмите Enter для продолжения...
```

Передача объекта по ссылке

В главе 6, “Создание функций”, была введена концепция передачи в функции аргументов простых типов по ссылке с помощью оператора `&`. Приведенная далее программа демонстрирует этот способ передачи для пользовательских объектов:

```
// PassObjRef - изменение значения объекта в функции
//              при передаче с использованием ссылки

#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  public:
    int    semesterHours;
    double gpa;
};

void someFn(Student& refS)
{
    refS.semesterHours = 10;
    refS.gpa           = 3.0;
    cout << "Значение refS.gpa = "
         << refS.gpa << "\n";
}

int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    s.gpa = 0.0;

    // Вывод значения s.gpa до вызова someFn()
    cout << "Значение s.gpa = " << s.gpa << "\n";

    // Передача существующего объекта
    cout << "Вызов someFn(Student&)\n";
    someFn(s);
    cout << "Возврат из someFn(Student&)\n";

    // Значение s.gpa теперь равно 3.0
    cout << "Значение s.gpa = " << s.gpa << "\n";

    // Ожидание, позволяющее пользователю увидеть результат
    // работы программы
    cout << "Нажмите Enter для продолжения..." << endl;
    cin.ignore(10, '\n');
    cin.get();
    return 0;
}
```

В этой программе в функцию `someFn()` передается не копия объекта, а ссылка на него. Вывод данной программы идентичен выводу предыдущей: изменения, внесенные функцией `someFn()` в `s`, сохраняются внутри `main()`.

Зачем использовать указатели и ссылки

Итак, передать объект в функцию можно разными способами. Но почему бы нам не ограничиться одним, простейшим способом — передачей по значению?

Один ответ мы уже получили, когда изучали способы передачи в этой главе: при передаче по значению нельзя изменить исходный объект, поскольку в функции вы работаете с копией объекта.

А вот и вторая причина: некоторые объекты могут оказаться *действительно очень большими*. Передача такого объекта по значению приводит к копированию большого объема информации в память функции.

При вызове из функции другой функции объект копируется вновь, и в результате нескольких вложенных вызовов вы получите десяток объектов в памяти и программу, работающую медленнее загрузки Windows.

Проблема на самом деле еще сложнее, чем описано здесь. В главе 17, “Копирующий/перемещающий конструктор”, вы убедитесь, что создание копии объекта представляет собой существенно более сложную задачу, чем простое копирование участка памяти из одного места в другое.

Передача указателя (или ссылки) — очень быстрый процесс. Размер указателя на большинстве современных платформ равен 4 байтам, независимо от того, каков размер объекта, на который этот указатель указывает.

Возврат к куче

Проблемы, возникающие при работе с указателями на простые переменные, распространяются и на указатели на объекты. В частности, необходимо гарантировать, что указатель ссылается на существующий корректный объект. Так, нельзя возвращать указатель на локально определенный объект, как это сделано в данном примере:

```
MyClass* myFunc()
{
    // Эта функция не будет работать правильно
    MyClass mc;
    MyClass* pMC = &mc;
    return pMC;
}
```

После возврата из `myFunc()` объект `mc` выходит из области видимости, а значит, указатель, который возвращает `myFunc()`, указывает на несуществующий объект.

Проблемы, связанные с возвратом памяти, которая выходит из области видимости, рассматривались в главе 9, “Второе знакомство с указателями в C++”.

Использование кучи позволяет решить эту проблему:

```
MyClass* myFunc()
{
    MyClass* pMC = new MyClass;
    return pMC;
}
```

Здесь память, выделенная из кучи, не возвращается в нее при выходе переменной `pMC` из области видимости.

С помощью кучи можно выделять память для объектов в самых разнообразных ситуациях, когда объект не должен потеряться при выходе некоторой переменной из области видимости. В этом случае программист сам отвечает как за выделение памяти для объекта, так и за освобождение ее, когда объект становится более не нужным.

Выделение из кучи памяти для кучи объектов

Можно также выделять из кучи память для массива объектов с применением следующего синтаксиса:

```
class MyClass
{
    public:
        int nValue;
};

void fn()
{
    MyClass* pMC = new MyClass[5]
    // Обращение к индивидуальным членам
    for (int i = 0; i < 5; i++)
    {
        pMC[i].nValue = i;
    }
    // Освобождение памяти - обратите внимание
    // на квадратные скобки
    delete[] pMC;
};
```

После выделения памяти `pMC` можно использовать, как и любой другой массив, с использованием синтаксиса `pMC[i]` для обращения к `i`-му объекту типа `MyClass`. Обратите внимание, что для освобождения памяти, выделенной для массива, используется несколько отличная форма оператора `delete[]` — с квадратными скобками.

Управление памятью вместо вас

Многие классы (в частности, описанные в главе 27, “Стандартная библиотека шаблонов”, контейнеры) работают с кучей вместо вас. Например, класс `string` хранит символьную строку в памяти, выделяемой из кучи. Авторы этого класса побеспокоились о том, чтобы память возвращалась в кучу, когда это требуется, так что вы можете безопасно писать следующий исходный текст:

```
string myFunc()
{
    string localString;
    localString << cin;
    return localString;
}
```

Объект `localString` при создании пользуется памятью из кучи, но корректно возвращает ее, выходя из области видимости в конце функции. (О том, как добиться такого чуда, вы узнаете из глав 16, "Аргументация конструирования", и 17, "Копирующий/перемещающий конструктор".)

Использование связанных списков

Связанный список является второй по распространенности структурой после массива. Каждый объект в связанном списке указывает на следующий, образуя цепочку в памяти. К связанному списку легко добавить еще один элемент — изменением указателя в последнем объекте списка. В этом заключается основное преимущество связанного списка — отсутствие необходимости задавать фиксированный размер на этапе компиляции: связанный список может уменьшаться и увеличиваться в зависимости от потребностей программы. Можно также сортировать члены связанного списка без перемещения объектов — простым изменением связей.

Цена этой гибкости — скорость работы со списком, поскольку обратиться к какому-то из элементов списка можно, только пройдя по всем предыдущим.

Связанный список, помимо изменяемого в процессе работы программы размера (что хорошо) и сложности доступа к произвольному объекту (что плохо), имеет еще одно свойство: интенсивное использование указателей. Это делает связанный список отличным средством для получения опыта в работе с указателями (что очень хорошо).

В стандартной библиотеке C++ имеется несколько типов списков. Вы увидите их в действии в главе 27, "Стандартная библиотека шаблонов"; однако при реализации собственного связанного списка вы получаете нечто бесценное, а именно — опыт.

Не всякий класс можно использовать для создания связанного списка. Связываемый класс объявляется так, как показано в приведенном ниже фрагменте:

```
class LinkableClass
{
    public:
        LinkableClass* pNext;
        // Прочие члены класса
};
```

Ключевым в этом классе является указатель `pNext` на объект класса `LinkableClass`. На первый взгляд, несколько необычно выглядит то, что класс содержит указатель сам на себя. В действительности в этом объявлении подразумевается, что каждый объект класса содержит указатель на другой объект этого же класса.

Связанный список похож на цепочку детей, идущих по улице, взявшись за руки. Указатель `pNext` соответствует в такой модели руке ребенка, которой он держит идущего за ним в цепочке.

Головной указатель является указателем типа `LinkableClass*`, и если использовать аналогию с цепочкой детей, держащихся за руки, то можно сказать, что учитель указывает на объект класса "ребенок" (любопытно отметить, что сам учитель не является ребенком — головной указатель не обязан иметь тип `LinkableClass`, он просто указатель на него).

Всегда инициализируйте указатели значением `nullptr`, указателем в никуда:

```
LinkableClass* pHead = nullptr;
```

В случае компиляторов, не соответствующих стандарту C++'11 и не реализующих `nullptr`, используйте значение `0` или соответствующую константу, определенную с помощью директивы `#define NULLPTR 0`.

```
LinkableClass* pHead = NULLPTR;
```

Чтобы увидеть, как связанные списки работают на практике, рассмотрим следующую функцию, которая добавляет переданный ей аргумент в начало списка:

```
void addHead(LinkableClass* pLC)
{
    pLC->pNext = pHead
    pHead = pLC;
}
```

Здесь после выполнения первой строки поле `pNext` указывает на первый член списка, а после второй строки заголовок списка указывает на добавленный элемент, что делает его первым элементом списка.

Другие операции над связанным списком

Добавление объекта в начало списка — самая простая операция со связанным списком. Хорошее представление о работе связанного списка дает процедура прохода по нему до конца списка:

```
// Проход по связанному списку
LinkableClass* pL = pHead;
while(pL)
{
    // Выполнение некоторых операций

    // Переход к следующему элементу
    pL = pL->pNext;
}
```

Сначала указатель `pL` инициализируется адресом первого объекта в списке (он хранится в переменной `pHead`). Затем программа входит в цикл `while`. Если указатель `pL` не нулевой, он указывает на некоторый объект `LinkableClass`. В этом цикле программа может выполнить те или иные действия над объектом.

После этого присваивание `pL = pL->pNext` "перемещает" указатель к следующему объекту списка. Если указатель становится нулевым, список исчерпан.

Программа `LinkedListData`

Программа `LinkedListData` использует связанный список для хранения списка объектов, содержащих имена людей. Программу очень легко расширить, добавив, например, номера социального страхования, рост или вес. Просто я старался сделать программу максимально простой:

```
// LinkedListData - хранение данных в связанном списке
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string.h>
using namespace std;

// NameDataSet - хранит имя человека (этот объект
//               можно легко расширить для хранения
//               другой информации)
class NameDataSet
{
  public:
    string sName;

    // Указатель на следующую запись в списке
    NameDataSet* pNext;
};

// Указатель на первую запись списка
NameDataSet* pHead = nullptr;

// Добавление нового члена в список
void add(NameDataSet* pNDS)
{
    pNDS->pNext = pHead;

    // Заголовок указывает на новую запись
    pHead = pNDS;
}

// getData - чтение имени
NameDataSet* getData()
{
    // Читаем имя
    string name;
    cout << "\nВведите имя: ";
    cin  >> name;

    // Если это имя — 'exit'...
    if (name == "exit")
    {
        // ...вернуть нулевое значение
        return 0;
    }

    // Новая запись для заполнения
    NameDataSet* pNDS = new NameDataSet;

    // Заполнение поля имени и обнуление указателя
    pNDS->sName = name;
    pNDS->pNext = 0;

    // Возврат адреса созданного объекта
    return pNDS;
}
```

```
int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "Читаем имена студентов\n"
         << "Введите 'exit' для выхода\n";

    // Создание объекта NameDataSet
    NameDataSet* pNDS;
    while (pNDS = getData())
    {
        // Добавление в конец списка
        add(pNDS);
    }

    // Итерация списка для вывода записей
    cout << "Записи:\n";
    for(NameDataSet *pIter = pHead;
            pIter; pIter = pIter->pNext)
    {
        // Вывод текущей записи
        cout << pIter->sName << endl;
    }

    // Ожидание, позволяющее пользователю увидеть результат
    // работы программы
    cout << "Нажмите Enter для продолжения..." << endl;
    cin.ignore(10, '\n');
    cin.get();
    return 0;
}
```

Несмотря на внушительную длину, программа `LinkedListData` относительно проста. Структура `NameDataSet` содержит поле для хранения имени персоны и ссылку на следующий в списке объект типа `NameDataSet`. Я уже упоминал, что в реальном приложении этот класс должен иметь и другие члены.

Для имени персоны я использовал класс `string`. Хотя все его методы не будут описаны до главы 27, “Стандартная библиотека шаблонов”, он гораздо проще в применении, чем строки с завершающим нулевым символом. В настоящее время класс `string` используется практически повсеместно и по сути, насколько это возможно, превращается во встроенный тип языка C++.

Функция `main()` начинается с циклического вызова функции `getData()`, которая считывает элемент `NameDataSet` с клавиатуры. Если пользователь вводит строку `exit`, `getData()` возвращает нулевой указатель. Функция `main()` вызывает функцию `add()`, чтобы добавить элемент, который вернула `getData()`, в конец связанного списка.

Если от пользователя больше не поступает элементов `NameDataSet`, функция `main()` выводит на экран все элементы списка.

Функция `getData()` ожидает ввода имени для записи его в соответствующее поле нового объекта. Если пользователь вводит в поле имени строку `exit`, функция возвращает нулевой указатель. В противном случае `getData()` считывает имя, после чего создает новый объект `NameDataSet`, заполняет его поле имени и обнуляет его указатель `pNext`.

Никогда не оставляйте связывающие указатели не инициализированными! Старая поговорка программистов гласит: “Не уверен — обнули”.

Наконец функция `getData()` возвращает функции `main()` адрес объекта.

Каждый объект, который возвращает функция `getData()`, добавляется в начало списка, на который указывает глобальная переменная-указатель `pHead`. Когда функция `getData()` возвращает нулевое значение, происходит выход из цикла `while`, после чего в следующем цикле осуществляется проход по списку с выводом информации о каждом элементе списка.

В этот раз я использовал цикл `for`, функционально эквивалентный использованному ранее циклу `while`. Цикл `for` инициализирует указатель `pIter` указателем на первый элемент списка присваиванием `pIter = pHead`. Затем выполняется проверка `pIter` на равенство нулю (что говорит об исчерпании списка). По окончании каждой итерации `pIter` переходит к следующему элементу списка с помощью присваивания `pIter = pIter->pNext`. Это стандартная процедура обхода списков любого типа:

```
Вывод программы имеет следующий вид.
Читаем имена студентов
Введите 'exit' для выхода

Введите имя: Игорь
Введите имя: Ира
Введите имя: Антон
Введите имя: Записи:
Антон
Ира
Игорь
Нажмите Enter для продолжения...
```

Вывод программы представляет собой введенные имена в обратном порядке. Это происходит потому, что добавление элементов выполняется в начало списка. Возможна вставка элементов в конец списка, однако эта задача посложнее, решенная в программе `LinkedListForward`, имеющейся на веб-сайте. Единственное отличие в ней — в функции `add()`, которая добавляет новые элементы в конец списка. Попытайтесь сами написать такую программу, прежде чем смотреть, как это сделал я.

Списки в стандартной библиотеке

Точно так, как ребенок должен научиться ходить перед тем, как ездить на автомобиле, считать — перед тем, как использовать калькулятор, программисту необходимо научиться писать программы, работающие со связанными списками, перед тем, как использовать классы списков, написанные другими. В главе 27, “Стандартная библиотека шаблонов”, будет описан связанный список из стандартной библиотеки C++.