**Динамические структуры данных: списки**

Ранее мы рассматривали программирование, связанное с обработкой только **статических данных. Статическими величинами** называются такие, память под которые выделяется во время компиляции и сохраняется в течение всей работы программы.

В языках программирования (Pascal, C, др.) существует и другой способ выделения памяти под данные, который называется **динамическим**. В этом случае память под величины отводится во время выполнения программы. Такие величины будем называть **динамическими**. Раздел оперативной памяти, распределяемый статически, называется **статической памятью**; динамически распределяемый раздел памяти называется **динамической памятью (динамически распределяемой памятью)**.

Использование динамических величин предоставляет программисту ряд дополнительных возможностей. Во-первых, подключение динамической памяти позволяет увеличить объем обрабатываемых данных. Во-вторых, если потребность в каких-то данных отпала до окончания программы, то занятую ими память можно освободить для другой информации. В-третьих, использование динамической памяти позволяет создавать структуры данных переменного размера.

Работа с динамическими величинами связана с использованием еще одного типа данных — **ссылочного типа**. Величины, имеющие ссылочный тип, называют **указателями**.

**Указатель** содержит адрес поля в динамической памяти, хранящего величину определенного типа. Сам указатель располагается в статической памяти.

**Адрес величины** — это номер первого байта поля памяти, в котором располагается величина. Размер поля однозначно определяется типом.

Далее будем более подробно обсуждать указатели и действия с ними в языке C/С++.

Величина ссылочного типа (указатель) описывается в разделе описания переменных следующим образом:

<имя типа> \*<идентификатор>;

Вот примеры описания указателей:

int \*Mas1;

float \*arr;

Здесь Mas1 — указатель на динамическую величину целого типа; arr — указатель на динамическую величину вещественного.

Сами динамические величины не требуют описания в программе, поскольку во время компиляции память под них не выделяется. Во время компиляции память выделяется только под статические величины. Указатели — это статические величины, поэтому они требуют описания.

Каким же образом происходит выделение памяти под динамическую величину? Память под динамическую величину, связанную с указателем, выделяется в результате выполнения стандартной процедуры NEW. Формат обращения к этой процедуре:

new <имя типа>;

Считается, что после выполнения этого оператора создана динамическая величина, имя которой имеет следующий вид:

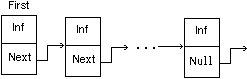
<имя динамической величины> = \*<указатель>

**Списки**

Обсудим вопрос о том, как в динамической памяти можно создать структуру данных переменного размера.

Разберём следующий пример. В процессе физического эксперимента многократно снимаются показания прибора (допустим, термометра) и записываются в компьютерную память для дальнейшей обработки. Заранее неизвестно, сколько будет произведено измерений.

Если для обработки таких данных не использовать внешнюю память (файлы), то разумно расположить их в динамической памяти. Во-первых, динамическая память позволяет хранить больший объем информации, чем статическая. А во-вторых, в динамической памяти эти числа можно организовать в связанный список, который не требует предварительного указания количества чисел, подобно массиву. Что же такое "связанный список"? Схематически он выглядит так:



Здесь Inf — информационная часть звена списка (*величина любого простого или структурированного типа, кроме файлового*), Next — указатель на следующее звено списка; First — указатель на заглавное звено списка.

Согласно определению, список располагается в динамически распределяемой памяти, в статической памяти хранится лишь указатель на заглавное звено. Структура, в отличие от массива, является действительно динамической: звенья создаются и удаляются по мере необходимости, в процессе выполнения программы.

Для объявления списка сделано исключение: указатель на звено списка объявляется раньше, чем само звено. В общем виде объявление выглядит так:

struct Item

{

BT Inf;

Item \*Next;

};

Здесь BT — некоторый базовый тип (Base Type) элементов списка.

Если указатель ссылается только на следующее звено списка (как показано на рисунке и в объявленной выше структуре), то такой список называют **однонаправленным**, если на следующее и предыдущее звенья — **двунаправленным списком**. Если указатель в последнем звене установлен не в null, а ссылается на заглавное звено списка, то такой список называется **кольцевым**. Кольцевыми могут быть и однонаправленные, и двунаправленные списки.

Более подробно рассмотрим работу со связанными списками на примере однонаправленного некольцевого списка.

Выделим типовые операции над списками:

* добавление звена в начало списка;
* удаление звена из начала списка;
* добавление звена в произвольное место списка, отличное от начала (например, после звена, указатель на которое задан);
* удаление звена из произвольного места списка, отличного от начала (например, после звена, указатель на которое задан);
* проверка, пуст ли список;
* очистка списка;
* печать списка.

Реализуем выделенный набор операций в виде модуля. Подключив этот модуль, можно решить большинство типовых задач на обработку списка. Пусть список объявлен так, как было описано выше. Первые четыре действия сначала реализуем отдельно, снабдив их иллюстрациями.

**1. Добавление звена в начало списка**

|  |  |
| --- | --- |
| Добавление звена в начало списка | {Процедура добавления звена в начало списка; в x содержится добавляемая информация}  Item \*addToHead(Item \*First, BT X)  {  Item \*Vsp;  Vsp = (Item \*) malloc(sizeof(Item));  Vsp->Inf=X;  Vsp->Next=First;  First=Vsp;  return First;  } |

**2. Удаление звена из начала списка**

|  |  |
| --- | --- |
| Удаление звена из начала списка | {Процедура удаления звена из начала списка;  в x содержится информация из удалённого звена}    Item \*removeFromHead(Item \*First)  {  Item \*Vsp;  Vsp=First->Next;  free(First);  return Vsp;  } |

**3. Добавление звена в произвольное место списка, отличное от начала (после звена, указатель на которое задан)**

|  |  |
| --- | --- |
| Добавление звена в произвольное место списка, отличное от начала (после звена, на которое ссылается указатель Pred) | {Процедура добавления звена в список после звена, на которое ссылается указатель Pred (predecessor - предыдущий);  в Х содержится информация для добавления}    Item \*addToList(Item \*Pred, BT X)  {  Item \*Vsp;  Vsp = (Item \*) malloc(sizeof(Item));  Vsp->Inf=X;  Vsp->Next=Pred->Next;  Pred->Next=Vsp;  return Vsp;  } |

**4. Удаление звена из произвольного места списка, отличного от начала (после звена, указатель на которое задан)**

|  |  |
| --- | --- |
| Удаление звена из произвольного места списка, отличного от начала (после звена, на которое ссылается указатель Pred) | {Процедура удаления звена из списка после звена, на которое ссылается указатель Pred;  в Х содержится информация из удалённого звена}    BT removeFromList(Item \*Pred)  {  BT X;  Item \*Vsp;  Vsp=Pred->Next;  Pred->Next=Pred->Next->Next;  X=Vsp->Inf;  free(Vsp);  return X;  } |

Приведём полный текст модуля list.cpp

|  |  |
| --- | --- |
|  | #include <iostream>  #include <stdlib.h>  typedef long BT;  struct Item  {  BT Inf;  Item \*Next;  };  Item \*addToHead(Item \*First, BT X)  {  Item \*Vsp;  Vsp = (Item \*) malloc(sizeof(Item));  Vsp->Inf = X;  Vsp->Next = First;  First = Vsp;  return First;  }  Item \*removeFromHead(Item \*First)  {  Item \*Vsp;  Vsp = First->Next;  free(First);  return Vsp;  }  Item \*addToList(Item \*Pred, BT X)  {  Item \*Vsp;  Vsp = (Item \*) malloc(sizeof(Item));  Vsp->Inf=X;  Vsp->Next = Pred->Next;  Pred->Next = Vsp;  return Vsp;  }  BT removeFromList(Item \*Pred)  {  BT X;  Item \*Vsp;  Vsp = Pred->Next;  Pred->Next = Pred->Next->Next;  X = Vsp->Inf;  free(Vsp);  return X;  }  void printList(Item \*First) // печать списка с начала  {  Item \*Vsp;  Vsp = First;  while (Vsp)  {  cout << Vsp->Inf << ' ';  Vsp = Vsp->Next;  }  cout << "\n";  }  int isEmpty(Item \*First) // проверка, пуст ли список  {  return !First;  }  Item \*clearList(Item \*First)  {  while (!isEmpty(First)) First = removeFromHead(First);  return First;  } |

**Пример.** Составить программу, которая на основе заданного списка формирует два других, помещая в первый из них положительные, а во второй — отрицательные элементы исходного списка.

При реализации алгоритма будем использовать подпрограммы разработанного модуля. Это существенно облегчает решение задачи.

#include "list.cpp"

using namespace std;

int main() {

Item \*S1, \*S2, \*S3, \*V1, \*V2, \*V3;

BT a;

int i, n;

S1 = NULL;

// создаём первый элемент

a = -100 + random() % 201;

S1 = addToHead(S1, a);

n = 1 + random() % 20;

// формируем список произвольной длины и выводим на печать

V1 = S1;

for (i = 2; i <= n; i++) {

a = -100 + random() % 201;

V1 = addToList(V1, a);

}

printList(S1);

V1 = S1;

S2 = NULL;

S3 = NULL;

while (V1)

{

if (V1 -> Inf > 0)

if (!S2) {

S2 = addToHead(S2, V1 -> Inf);

V2 = S2;

} else {

addToList(V2, V1 -> Inf);

V2 = V2 -> Next;

};

if (V1 -> Inf < 0)

if (!S3) {

S3 = addToHead(S3, V1 -> Inf);

V3 = S3;

} else {

addToList(V3, V1 -> Inf);

V3 = V3 -> Next;

};

V1 = V1 -> Next;

}

cout << "Результирующий список из положительных элементов: \n";

printList(S2);

cout << "Результирующий список из отрицательных элементов: \n";

printList(S3);

S1 = clearList(S1);

S2 = clearList(S2);

S3 = clearList(S3);

return 0;

}