基于 Linux 的 C++

第八讲　链表与程序抽象

提　纲

数据抽象

链　表

函数指针

抽象链表

数据抽象

- 数据抽象的目的与意义
- 结构化数据类型的性质
- 数据封装
- 信息隐藏
- 抽象数据类型

数据抽象的目的与意义

数据对象

信息缺失：程序中的数据对象只有地址和值，没有数据类型、数据解释及数据意义等信息

解决手段：**抽象**

数据的表示：注释、有意义的数据对象名称

数据的功能：描述可以在数据上工作的操作集

数据的功能比表示更重要

例：程序员更关心整数的运算而不是计算机如何存储整数

结构化数据类型的性质

类　型

细节由用户自定义，语言仅提供定义手段

成　员

结构化数据类型的子数据对象

成员类型

每个成员具有确切的类型

成员数目

部分结构化数据类型可变，部分固定

成员组织

成员组织结构（线性结构或非线性结构）必须显式定义

操作集

可以在数据上进行的操作集合

数据封装

数据封装：将数据结构的细节隐藏起来

实现方式：分别实现访问数据成员的存取函数

数据封装示例

```
struct DYNINTS{
  unsigned int capacity;
  unsigned int count;
  int * items;
  bool modified;
};
unsigned int DiGetCount( DYNINTS* a )
{
  if( !a ){ cout << "DiGetCount: Parameter illegal." << endl;  exit(1);  }
  return a->count;
}
```

信息隐藏

数据封装的问题

只要将结构体类型定义在头文件中，库的使用者就可以看到该定义，并按照成员格式直接访问，而不调用存取函数

解决方法

将结构体类型的具体细节定义在源文件中，所有针对该类型量的操作都只能通过函数接口来进行，从而隐藏实现细节

信息隐藏示例

```
/* 头文件“dynarray.h”*/
struct DYNINTS;  typedef struct DYNINTS * PDYNINTS;
/* 源文件“dynarray.cpp”*/
struct DYNINTS{
  unsigned int capacity;  unsigned int count;  int * items;  bool modified;
};
```

抽象数据类型

设计能够存储二维平面上点的抽象数据类型

```
/*  点库接口“point.h”*/

struct POINT;
typedef struct POINT * PPOINT;

PPOINT PtCreate( int x, int y );
void PtDestroy( PPOINT point );
void PtGetValue( POINT point, int * x, int * y );
void PtSetValue( PPOINT point, int x, int y );
bool PtCompare( PPOINT point1, PPOINT point2 );
char * PtTransformIntoString( PPOINT point );
void PtPrint( PPOINT point );
```

抽象数据类型

```
/*  点库实现“point.cpp”*/
#include <cstdio>
#include <cstring>
#include <iostream>
#include "point.h"
using namespace std;
static char* DuplicateString( const char* s );
struct POINT{  int x, y;  };

PPOINT PtCreate( int x, int y )
{
  PPOINT t = new POINT;    t->x = x;    t->y = y;    return t;
}
void PtDestroy( PPOINT point )
{
  if( point ){  delete  point;  }
}
```

抽象数据类型

```
void PtGetValue( PPOINT point, int * x, int * y )
{
  if( point ){  if( x )    *x = point->x;    if( y )    *y = point->y;  }
}
void PtSetValue( PPOINT point, int x, int y )
{
  if( point ){  point->x = x;    point->y = y;  }
}
bool PtCompare( PPOINT point1, PPOINT point2 )
{
  if( !point1 || !point2 ){  cout << "PtCompare: Parameter(s) illegal." << endl;  exit(1);  }
  return ( point1->x == point2->x ) && ( point1->y == point2->y );
}
void PtPrint( PPOINT point )
{
  if( point )    printf( "(%d,%d)", point->x, point->y );
  else    printf( "NULL" );
}
```

抽象数据类型

```
char * PtTransformIntoString( PPOINT point )
{
  char buf[BUFSIZ];
  if( point ){
    sprintf( buf, "(%d,%d)", point->x, point->y );
    return DuplicateString( buf );
  }
  else  return "NULL";
}

char* DuplicateString( const char* s )
{
  unsigned int n = strlen(s);
  char* t = new char[n+1];
  for( int i=0; i<n; i++)
    t[i] = s[i];
  t[n] = '\0';
  return t;
}
```

链　表

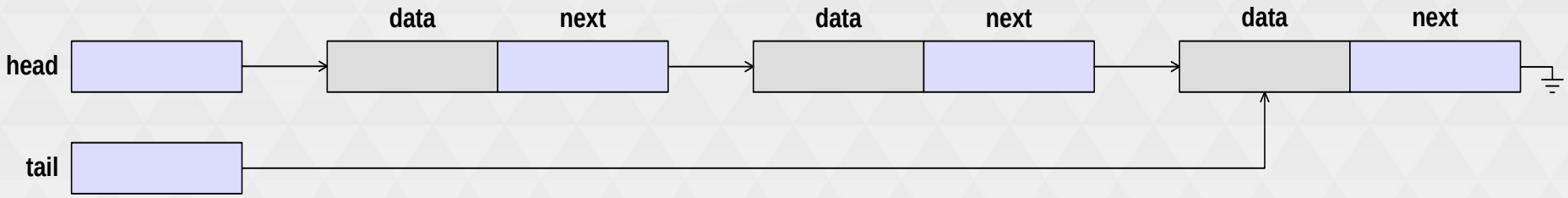
链表的意义与性质

存储顺序访问的数据对象集

数据对象占用的存储空间总是动态分配的

链表的定义

元素序列，每个元素与前后元素相链接



结点：链表中的元素

表头、表尾：链表的头尾结点

头指针、尾指针：指向表头、表尾的指针

链表数据结构

链表结点：使用结构体类型表示

至少包含两个域：结点数据域与链接指针域

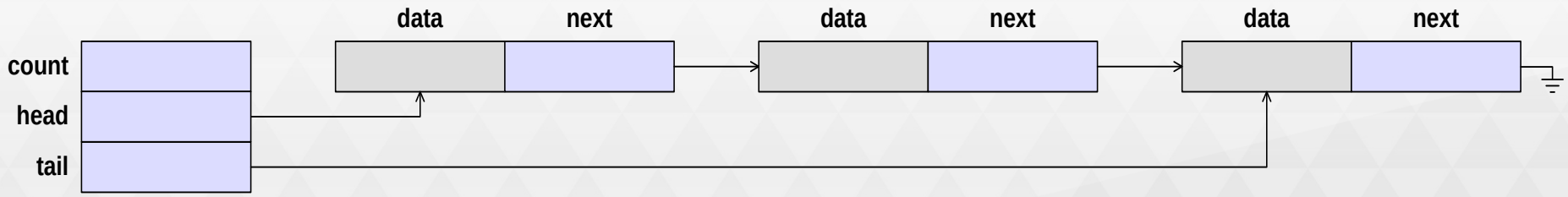
```
struct NODE;  typedef struct NODE * PNODE;
struct NODE{
    PPOINT data;    /* 当前结点的存储数据  */
    PNODE next;    /* 指向下一结点，表尾此域为 NULL */
};
```

链表结构：封装结点表示的细节

```
struct LIST;  typedef struct LIST * PLIST;
struct LIST{
    unsigned int count;    /* 链表中包含的结点数目 */
    PNODE head, tail;    /* 链表头尾指针 */
};
```

链表数据结构

标准链表图例



特别说明

结点总是动态分配内存的，所以结点逻辑上连续，物理上地址空间并不一定连续

时刻注意维护链表的完整性：一旦头指针 head 失去链表表头地址，整个链表就会丢失；任一结点 next 域失去下一结点地址，后续结点就会全部丢失

单向链表、双向链表、循环链表、双向循环链表

抽象链表接口

设计能够处理点数据类型的抽象链表接口

```
#include "point.h"

struct LIST;
typedef struct LIST * PLIST;

PLIST LlCreate();
void LlDestroy( PLIST list );
void LlAppend( PLIST list, PPOINT point );
void LlInsert( PLIST list, PPOINT point, unsigned int pos );
void LlDelete( PLIST list, unsigned int pos );
void LlClear( PLIST list );
void LlTraverse( PLIST list );
bool LlSearch( PLIST list, PPOINT point );
unsigned int LlGetCount( PLIST list );
bool LlIsEmpty( PLIST list );
```

抽象链表实现

链表的构造与销毁
结点的追加
结点的插入
结点的删除
结点的遍历
结点的查找

链表的构造与销毁

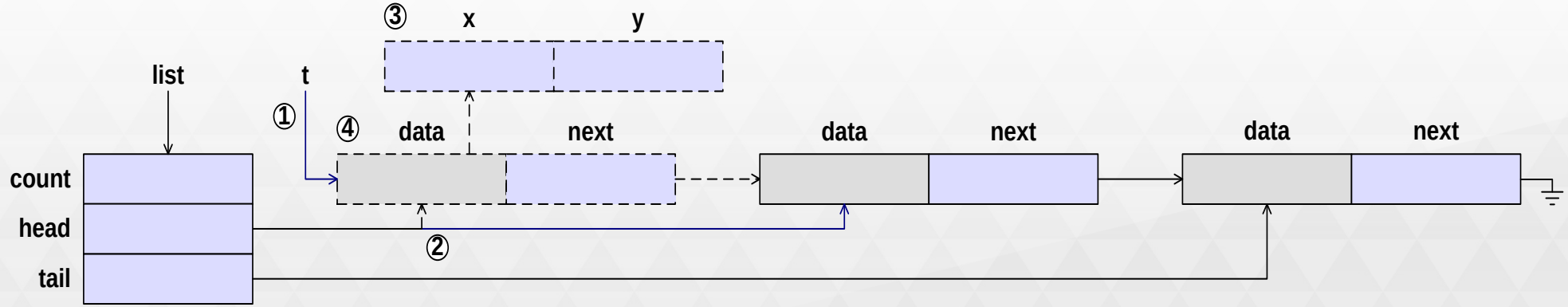
编写函数，实现链表的构造与销毁操作

```
PLIST LlCreate()
{
  PLIST p = new LIST;
  p->count = 0;
  p->head = NULL;
  p->tail = NULL;
  return p;
}
void LlDestroy( PLIST list )
{
  if( list )
  {
    LlClear( list );
    delete list;
  }
}
```

链表的构造与销毁

```
void LlClear( PLIST list )
{
  if( !list )
  {
    cout << "LlClear: Parameter illegal." << endl;
    exit(1);
  }
  while( list->head )
  {
    PNODE t = list->head;
    list->head = t->next;
    PtDestroy( t->data );
    delete t;
    list->count--;
  }
  list->tail = NULL;
}
```

表头结点的删除



操作步骤

设置临时指针 t，使其指向链表头结点

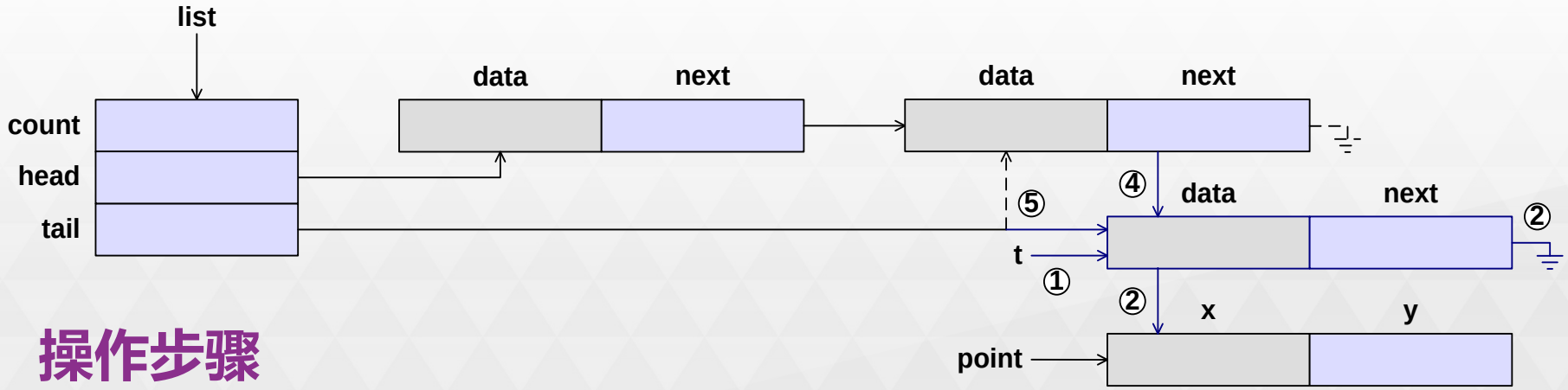
将链表头结点设置为 t 的后继结点

删除原头结点 data 域所指向的目标数据对象

删除 t 所指向的结点

递减链表结点数目

结点的追加



操作步骤

动态构造一个新结点，用 t 指向它

使 t 的 data 域指向 point 参数指向的目标数据对象，next 域为NULL

如果链表的 head 域为 NULL，则说明当前链表中没有任何结点，将此结点作为链表惟一结点添加到链表中，此时简单将链表的 head 域与 tail 域设为 t 即可

否则，将当前尾结点的 next 域设为 t，即使其指向新结点

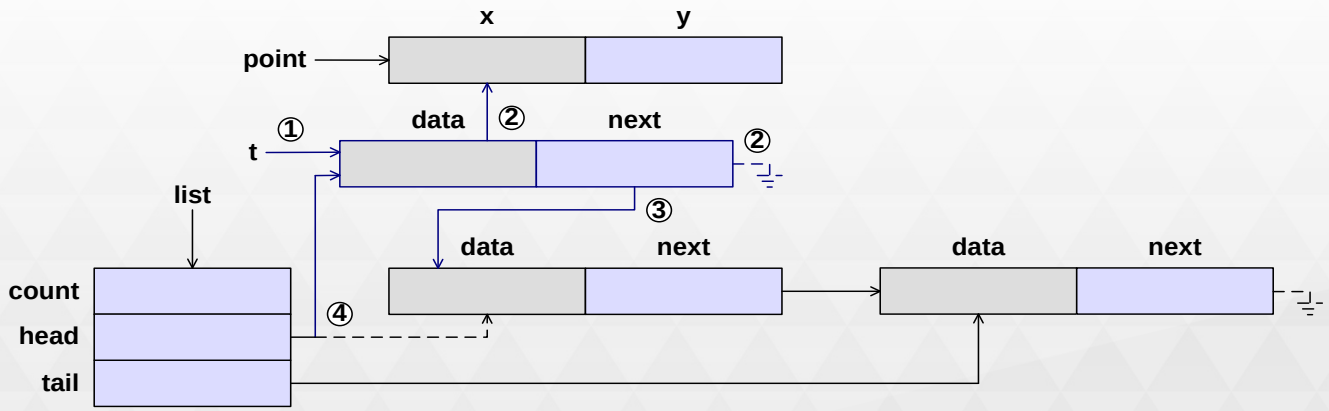
将链表的 tail 域设为 t，即将新结点作为链表尾结点

递增链表结点数目

结点的追加

```
void LlAppend( PLIST list, PPOINT point )
{
  PNODE t = new NODE;
  if( !list || !point ){  cout << "LlAppend: Parameter illegal." << endl;  exit(1);  }
  t->data = point;
  t->next = NULL;
  if( !list->head )
  {
    list->head = t;
    list->tail = t;
  }
  else
  {
    list->tail->next = t;
    list->tail = t;
  }
  list->count++;
}
```

结点的插入



表头插入的操作步骤

动态构造一个新结点，用 t 指向它

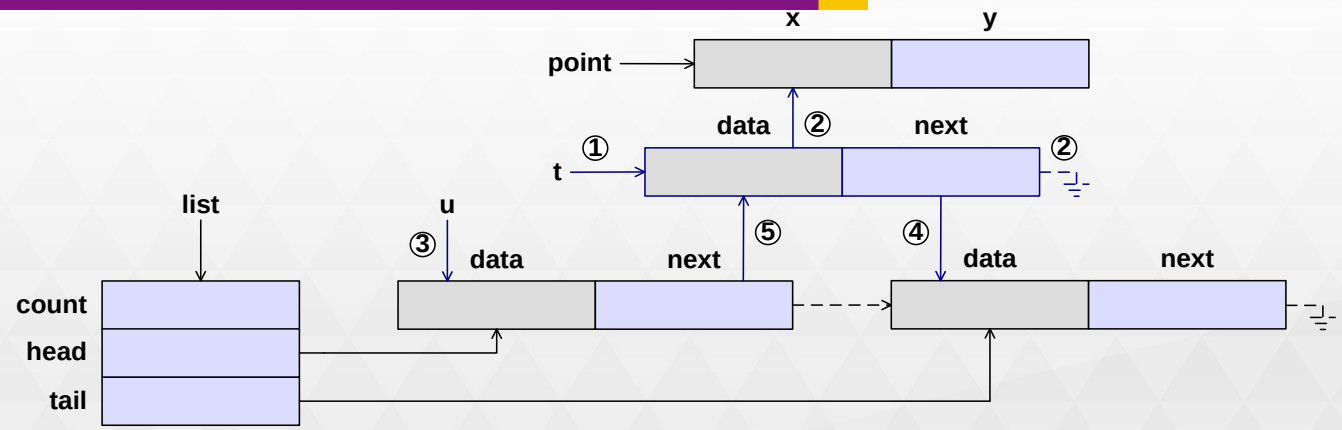
使 t 的 data 域指向 point 指向的目标数据对象，next 域为NULL

将 t 的 next 域设为 list 的 head 的值，即使得原链表首结点链接到 t 所指向的结点之后

修改链表首结点指针，使其指向新结点

递增链表的结点数目

结点的插入



表中插入的操作步骤

动态构造一个新结点，用 t 指向它

使 t 的 data 域指向 point 指向的目标数据对象，next 域为 NULL

从表头开始向后查找待插入位置的前一结点，用 u 指向它，例如若插入位置为 1，则用 u 指向 0 号结点

将 t 的 next 域设为 u 的 next 的值，即使得原链表中位置 pos 处的结点链接到 t 所指向的结点之后

将 u 的 next 域设为 t，即将 t 指向的结点链接到 u 指向的结点之后递增链表的结点数目

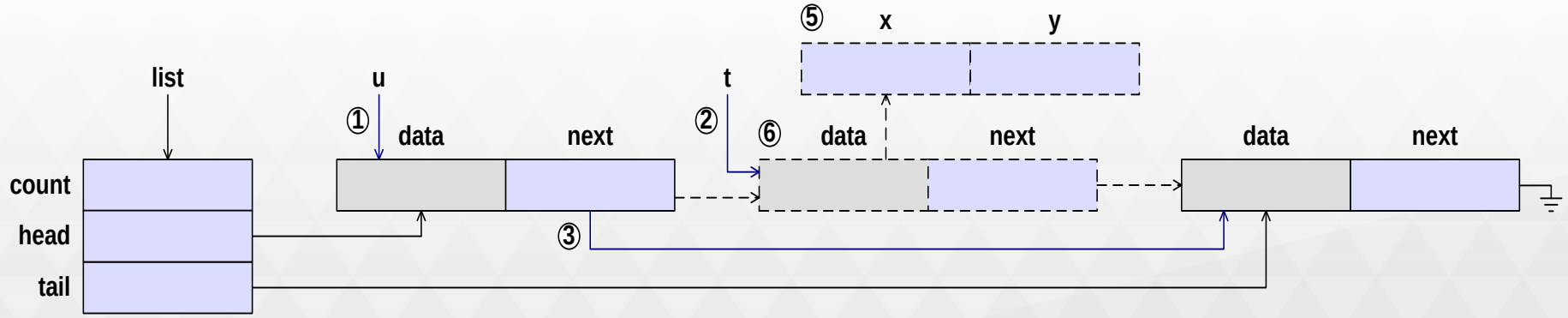
结点的插入

```
void LlInsert( PLIST list, PPOINT point, unsigned int pos )
{
  if( !list || !point )
  {
    cout << "LlInsert: Parameter illegal." << endl;
    exit(1);
  }
  if( pos < list->count )
  {
    PNODE t = new NODE;
    t->data = point;
    t->next = NULL;
    if( pos == 0 )
    {
      t->next = list->head;
      list->head = t;
    }
```

结点的插入

```
 else
    {
      unsigned int i;
      PNODE u = list->head;
      for( i = 0; i < pos - 1; ++i )
        u = u->next;
      t->next = u->next;
      u->next = t;
    }
    list->count++;
  }
  else
    LlAppend( list, point );
}
```

结点的删除



表中或表尾删除的操作步骤

使用临时指针 u 保存待删除结点前一结点的地址

t 保存待删除结点的地址

将 t 的 next 域赋给 u 的 next 域，这保证 u 跳过 t 指向下一结点

若 t 的 next 域不再指向其他结点（t 指向的结点本身就是链表尾结点）则将链表尾结点设为 u

释放 t 的 data 域所指向的目标数据对象

释放 t 所指向的结点数据对象

递减链表的结点个数

结点的删除

```
void LlDelete( PLIST list, unsigned int pos )
{
  if( !list )
  {
    cout << "LlDelete: Parameter illegal." << endl;
    exit(1);
  }
  if( list->count == 0 )
    return;
  if( pos == 0 )
  {
    PNODE t = list->head;
    list->head = t->next;
    if( !t->next )
      list->tail = NULL;
    PtDestroy( t->data );
    delete t;
    list->count--;
  }
```

结点的删除

```
else if( pos < list->count )
 {
   unsigned int i;
   PNODE u = list->head, t;
   for( i = 0; i < pos - 1; ++i )
     u = u->next;
   t = u->next;
   u->next = t->next;
   if( !t->next )
     list->tail = u;
   PtDestroy( t->data );
   delete t;
   list->count--;
 }
}
```

链表的遍历

编写函数，遍历链表，调用 PtTransformIntoString 函数输出结点数据，相邻结点使用“->”连接

```
void LlTraverse( PLIST list )
{
  PNODE t = list->head;
  if( !list )
  {
    cout << "LlTraverse: Parameter illegal." << endl;
    exit(1);
  }
  while( t )
  {
    cout << PtTransformIntoString(t->data) << " -> ";
    t = t->next;
  }
  cout << "NULL\n";
}
```

链表的查找

编写函数，在链表中查找特定点 point 是否存在

```
bool LlSearch( PLIST list, PPOINT point )
{
  PNODE t = list->head;
  if( !list || !point )
  {
    cout << "LlSearch: Parameter illegal." << endl;
    exit(1);
  }
  while( t )
  {
    if( PtCompare( t->data, point ) )
      return true;
    t = t->next;
  }
  return false;
}
```

链表小结

链表的优点

插入和删除操作非常容易，不需要移动数据，只需要修改链表结点指针

与数组比较：数组插入和删除元素操作则需要移动数组元素，效率很低

链表的缺点

只能顺序访问，要访问某个结点，必须从前向后查找到该结点，不能直接访问

链表设计中存在的问题

链表要存储点数据结构，就必须了解点库的接口；如果要存储其他数据，就必须重新实现链表

若要存储目前还没有实现的数据结构，怎么办？

函数指针

函数指针的目的与意义：抽象数据与抽象代码

数据与算法的对立统一

函数的地址：函数入口位置，将该数值作为数据保存起来，就可以通过特殊手段调用该函数

typedef void * ADT; typedef const void * CADT;

将链表所要存储的结点数据对象抽象成通用类型，不允许在链表库中出现与点数据结构相关的任何东西

函数指针的定义

函数指针的使用

函数指针类型

函数指针的定义

函数指针的定义格式

数据类型 (* 函数指针数据对象名称)(形式参数列表);

示例：char * (* as_string)(ADT object);

函数指针变量的赋值

as_string 作为变量可以指向任何带有一个 ADT 类型参数的返回值为 char * 类型的函数

函数指针变量可以像普通变量一样赋值

函数指针数据对象名称 = 函数名称;

```
char * DoTransformObjectIntoString( ADT object )
{  return PtTransformIntoString( (PPOINT)object );  }
as_string = DoTransformObjectIntoString;
```

函数指针的使用

通过函数指针调用函数

函数指针被赋值后，即指向实际函数的入口地址

通过函数指针可以直接调用它所指向的函数

调用示例：

```
char * returned_value;
PPOINT pt = PtCreate( 10, 20 );
as_string = DoTransformObjectIntoString;
returned_value = as_string( (ADT)pt );
```

要区分函数指针调用和函数直接调用，使用下述格式调用函数指针指向的函数：

```
returned_value = ( *as_string )( (ADT)pt );
```

函数指针的使用

设计程序，随机生成 8 个 10~99 之间的整数，调用 stdlib 库的 qsort 函数对其进行排序

qsort 函数原型

```
void qsort( void * base, unsigned int number_of_elements,
    unsigned int size_of_elements,
    int ( * compare )( const void *, const void * ) );
```

调用时需按照下述格式实现自己的比较函数

```
int ( * compare )( const void *, const void * );
```

比较函数示例

```
int MyCompareFunc( const void * e1, const void * e2 );
```

比较函数必须返回正负值（一般为正负 1）或 0 ，规则按照题目要求自定义

函数指针的使用：main.cpp

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

#include "arrmanip.h"

#define NUMBER_OF_ELEMENTS 8

int DoCompareObject( const void * e1, const void * e2 );
```

函数指针的使用：main.cpp

```
int main()
{
  int a[NUMBER_OF_ELEMENTS];
  GenerateIntegers( a, NUMBER_OF_ELEMENTS );
  cout << "Array generated at random as follows: \n";
  PrintIntegers( a, NUMBER_OF_ELEMENTS );
  qsort( a, NUMBER_OF_ELEMENTS, sizeof(int), DoCompareObject );
  cout << "After sorted: \n";
  PrintIntegers( a, NUMBER_OF_ELEMENTS );
  return 0;
}
int DoCompareObject( const void * e1, const void * e2 )
{
  return CompareInteger( *(const int *)e1, *(const int *)e2 );
}
```

函数指针的使用：arrmanip.h

```
void GenerateIntegers( int a[], unsigned int n );

int CompareInteger( int x, int y );

void PrintIntegers( int a[], unsigned int n );
```

函数指针：arrmanip.cpp

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

#include "random.h"
#include "arrmanip.h"

static const unsigned int lower_bound = 10;
static const unsigned int upper_bound = 99;

void GenerateIntegers( int a[], unsigned int n )
{
  unsigned int i;
  Randomize();
  for( i = 0; i < n; i++ )
    a[i] = GenerateRandomNumber( lower_bound, upper_bound );
}
```

函数指针：arrmanip.cpp

```
int CompareInteger( int x, int y )
{
  if( x > y )
    return 1;
  else if( x == y )
    return 0;
  else
    return -1;
}

void PrintIntegers( int a[], unsigned int n )
{
  unsigned int i;
  for( i = 0; i < n; i++ )
    cout << setw(3) << a[i];
  cout << endl;
}
```

函数指针的使用

函数指针的赋值

同类型函数指针可以赋值，不同类型则不能赋值

如何确定函数指针类型是否相同：函数参数与返回值不完全相同

函数指针类型：用于区分不同类型的函数指针

```
typedef int ( * COMPARE_OBJECT )( const void * e1, const void * e2 );
```

前面添加 typedef 关键字，保证 COMPARE_OBJECT 为函数指针类型，而不是函数指针变量

可以像普通类型一样使用函数指针类型定义变量：

```
COMPARE_OBJECT compare = DoCompareObject;
```

qsort 函数的简明书写方法

```
void qsort( void * base, unsigned int number_of_elements, unsigned int
size_of_elements, COMPARE_OBJECT compare );
```

抽象链表

回调函数

- 允许通过函数指针调用未来才会实现的代码
- 回调函数：依赖后续设计才能确定的被调函数
- 示例：DoCompareObject

回调函数参数

- 回调函数与主调函数的信息交互：附加信息

数据对象的存储与删除

- 删除链表结点时，其中的目标数据对象是否需要删除？
- 如果链表结点存储的是指针，就需要删除；否则不需要
- 设计抽象链表时，并不了解结点实际存储的数据是否为指针，因而无法确定结点数据操作逻辑

回调函数

编写函数，遍历链表，结点数据的具体操作方法目前未知，由未来的回调函数提供

```
typedef void ( * MANIPULATE_OBJECT )( ADT e );
void LlTraverse( PLIST list, MANIPULATE_OBJECT manipulate )
{
  PNODE t = list->head;
  if( !list )
  {
    cout << "LlTraverse: Parameter illegal." << endl;
    exit(1);
  }
  while( t )
  {
    if( manipulate )    /* 通过函数指针调用实际函数操纵目标数据对象 */
      ( *manipulate )( t->data );
    t = t->next;
  }
}
```

回调函数参数

```
typedef void ( * MANIPULATE_OBJECT )( ADT e, ADT tag );
/* 链表遍历函数 */
void LlTraverse( PLIST list, MANIPULATE_OBJECT manipulate, ADT tag )
{
  PNODE t = list->head;
  if( !list )
  {
    cout << "LlTraverse: Parameter illegal." << endl;
    exit(1);
  }
  while( t )
  {
    if( manipulate )
      ( *manipulate )( t->data, tag );
    t = t->next;
  }
}
```

回调函数参数

```
/* 点数据到字符串的转换函数，最终程序员任意定义 */
/* 参数 format 表示点数据对象的转换格式 */
/* 其中只能包含两个格式码 %d，其他内容任意 */
/* 例如格式“(%d,%d)”或“[%4d, %4d]”等 */
char * PtTransformIntoString( const char * format, PPOINT point )
{
  char buf[BUFSIZ];
  if( point )
  {
    sprintf( buf, format, point->x, point->y );
    return DuplicateString( buf );
  }
  else
    return "NULL";
}
```

回调函数参数

```
/*  回调函数 DoPrintObject */
void DoPrintObject( ADT e, ADT tag )
{
  printf( PtTransformIntoString( (const char *)tag, (PPOINT)e ) );
  printf( " -> " );
}

/*  回调函数参数的意义 */
/*  调用遍历函数时将点数据的输出格式传递给遍历函数 */
/*  再由遍历函数传递给回调 */
LlTraverse( list, DoPrintObject, "(%d,%d)" );
```

回调函数参数

回调函数参数的重要意义

- 程序的参与者：抽象链表的设计者、点数据结构的设计者、最终使用前两者的第三方程序员
- 优势：三者完全不了解其他人的实现细节

容器与容器中的对象

- 容器：能够容纳其他数据对象集合的东西
- 抽象链表：容器；点数据结构：容器中的数据对象
- 两者完全无关，即容器与容器中容纳的数据对象完全独立
- **抽象链表事实上可以存储任意类型的数据对象**

数据对象的存储与删除

链表中 data 域是否为指针？

data 域是否真正指向一个存在的目标数据对象？

如果结点被删除，data 域指向的目标数据对象是否需要删除？

如果需要删除，如何删除？

抽象链表的设计者能够完成此删除任务吗？

如果不能，怎么处理它？

数据对象的存储与删除

```
typedef void ( * DESTROY_OBJECT )( ADT e );
void LlDelete( PLIST list, unsigned int pos, DESTROY_OBJECT destroy )
{
  // ......
  if( pos == 0 )
  {
    // ......
    if( destroy )
      ( *destroy )( t->data );
    // ......
  }
  else if( pos < list->count )
  {
    // ......
    if( destroy )
      ( *destroy )( t->data );
    // ......
  }
}
```

数据对象的存储与删除

若需要删除目标数据对象，实现下述代码

```
void DoDestroyObject( ADT e )
{
  delete (PPOINT)e;
}

/* 调用 DoDestroyObject 函数释放 data 域指向的存储空间 */
LlDelete( list, 1, DoDestroyObject );
```

若不需要删除目标数据对象，实现下述代码

```
LlDelete( list, 1, NULL );
```

数据对象的存储与删除

设计不依赖所存储的具体数据类型的抽象链表

```
typedef struct LIST * PLIST;

typedef int ( * COMPARE_OBJECT )( CADT e1, CADT e2 );
typedef void ( * DESTROY_OBJECT )( ADT e );
typedef void ( * MANIPULATE_OBJECT )( ADT e, ADT tag );

PLIST LlCreate();
void LlDestroy( PLIST list, DESTROY_OBJECT destroy );
void LlAppend( PLIST list, ADT object );
void LlInsert( PLIST list, ADT object, unsigned int pos );
void LlDelete( PLIST list, unsigned int pos, DESTROY_OBJECT destroy );
void LlClear( PLIST list, DESTROY_OBJECT destroy );
void LlTraverse( PLIST list, MANIPULATE_OBJECT manipulate, ADT tag );
bool LlSearch( PLIST list, ADT object, COMPARE_OBJECT compare );
unsigned int LlGetCount( PLIST list );
bool LlIsEmpty( PLIST list );
```

编程实践

8.1 实现动态数组库。

8.2 实现抽象链表库。