基于 Linux 的 C++

第七讲 指针与引用

提　纲

指　针

指针的基本概念、指针运算、指针与函数、指针与数组、指针与结构体

字符串

字符数组、字符指针、字符串整体、C标准字符串库、C++字符串类

动态存储管理

C格式：malloc/free

C++格式：new/delete

引　用

指　针

数据对象的地址与值

- 地址：数据对象的存储位置在计算机中的编号
- 值：在该位置处存储的内容
- 地址与值是辩证统一的关系

指针变量的定义与使用

指针与函数

指针与复合数据类型

- 指针运算
- 指针与数组
- 指针与结构体

指针的定义与使用

指针的定义格式

格式：目标数据对象类型 * 指针变量名称;

例一：定义 p 为指向整数的指针：int * p;

例二：定义 p 为指向结构体类型的指针：

struct POINT{ int x, y; }; POINT * p;

多个指针变量的定义

例三：int * p, * q;

例四：typedef int * PINT; PINT p, q;

指针变量的存储布局

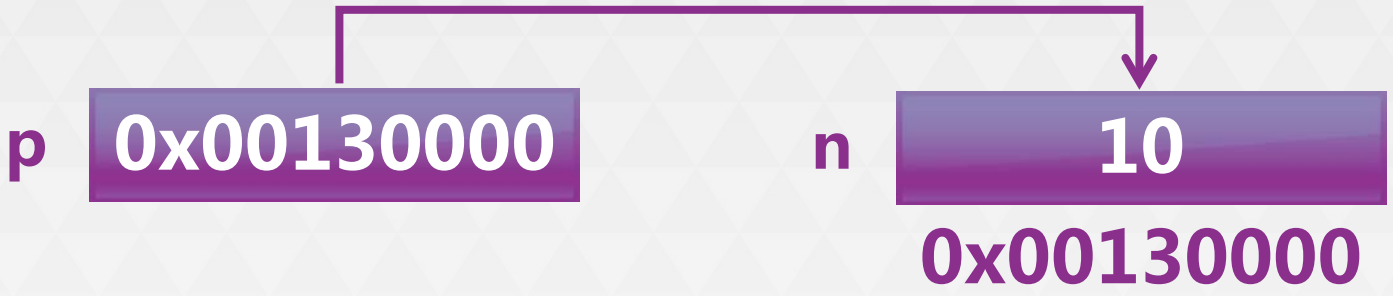
指针数据对象（变量）与目标数据对象（变量）

仅定义指针变量，未初始化

例一：int * p;

定义指针变量，并使其指向某个目标变量

例二：int n = 10; int * p = &n;



指针变量的存储布局

定义指针变量，并使其指向数组首元素

例三：int a[8] = {1, 2, 3, 4, 5, 6, 7, 8}; int * p = a;

指针变量的赋值

指针变量可以像普通变量一样赋值

示例：int n = 10; int * p = &n, * q; q = p;

两个指针指向同一个目标数据对象

取址操作符

取址操作符“&”

获取数据对象的地址，可将结果赋给指针变量

示例：int n = 10; int * p; p = &n; int * q; q = p;

引领操作符

引领操作符

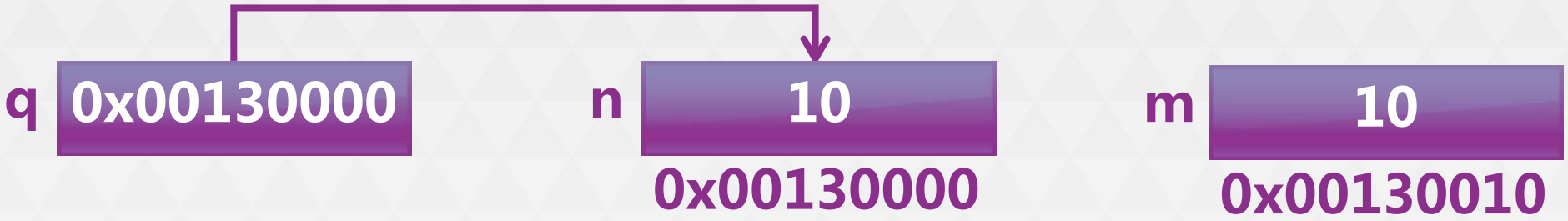
获取指针所指向的目标数据对象

例一：int m, n = 10; int * q = &n; m = *q;

使得 m 为 10

例二（接上例）：*q = 1;

使得 n 为 1



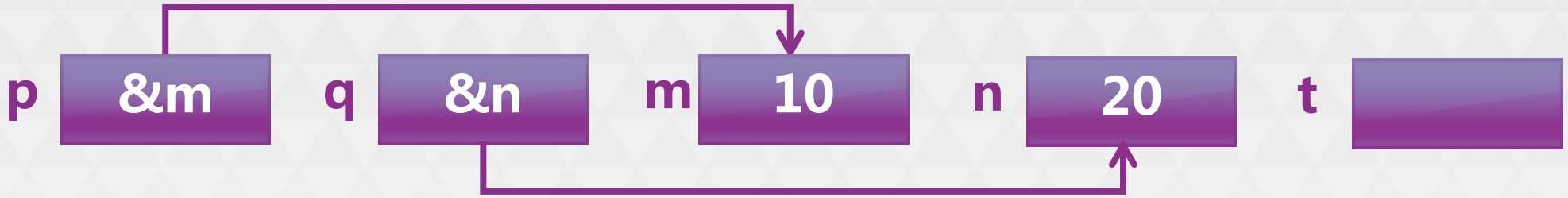
整数互换程序

编写程序，使用指针互换两个整数的值

```
1       #include <iostream>
2       using namespace std;
3       int main(){
4         int m = 10, n = 20, t;
5         int *p = &m, *q = &n;
6         cout << "m: " << m << "; n: " << n << endl;
7         t = *p;
8         *p = *q;
9         *q = t;
10        cout << "m: " << m << "; n: " << n << endl;
11        return 0;
12      }
```

目标数据对象的操作

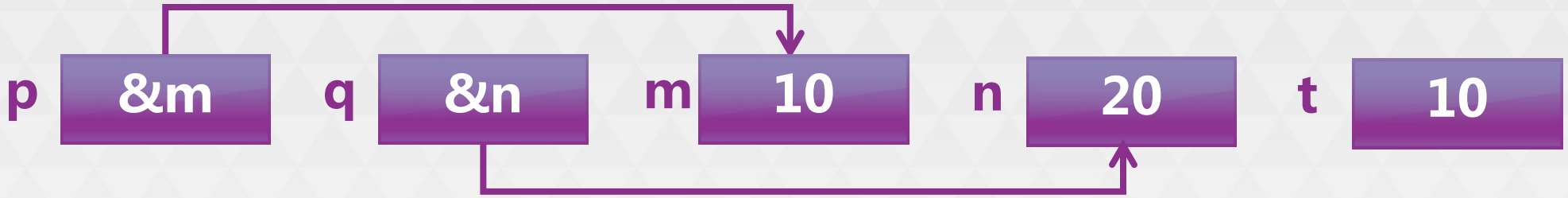
```
t = *p;
```



第七条语句执行前

目标数据对象的操作

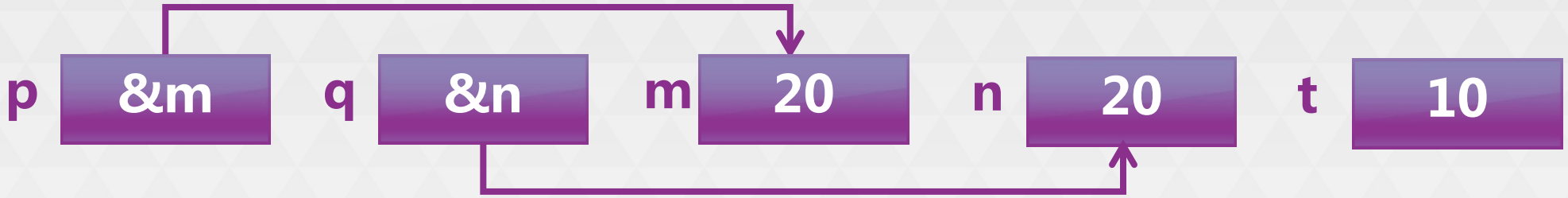
```
t = *p;
```



第七条语句执行后

目标数据对象的操作

```
t = *p;
*p = *q;
```

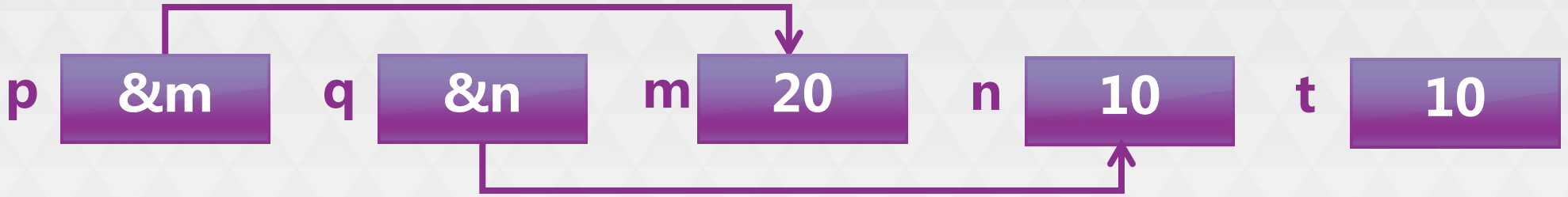


第八条语句执行后

目标数据对象的操作

```
t = *p;
*p = *q;
*q = t;
```

互换 m、n 的值，
p、q 的指向没有变化



第九条语句执行后

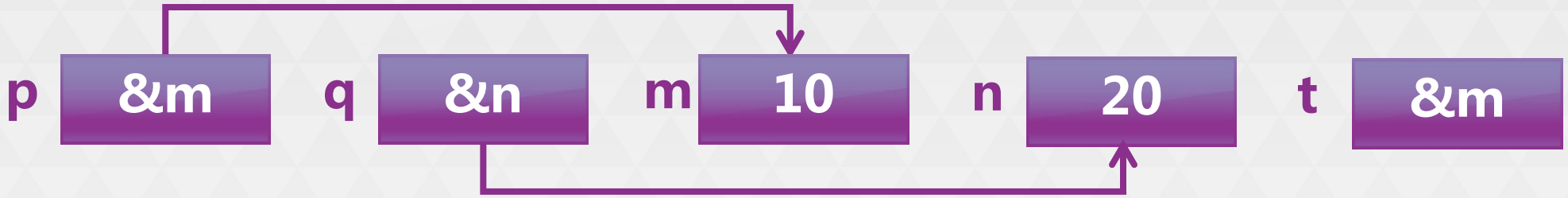
整数互换程序

编写程序，使用指针互换两个整数的值

```
1       #include <iostream>
2       using namespace std;
3       int main(){
4         int m = 10, n = 20;
5         int *p = &m, *q = &n, *t;
6         cout << "m: " << m << "; n: " << n << endl;
7         t = p;
8         p = q;
9         q = t;
10        cout << "m: " << m << "; n: " << n << endl;
11        return 0;
12      }
```

目标数据对象的操作

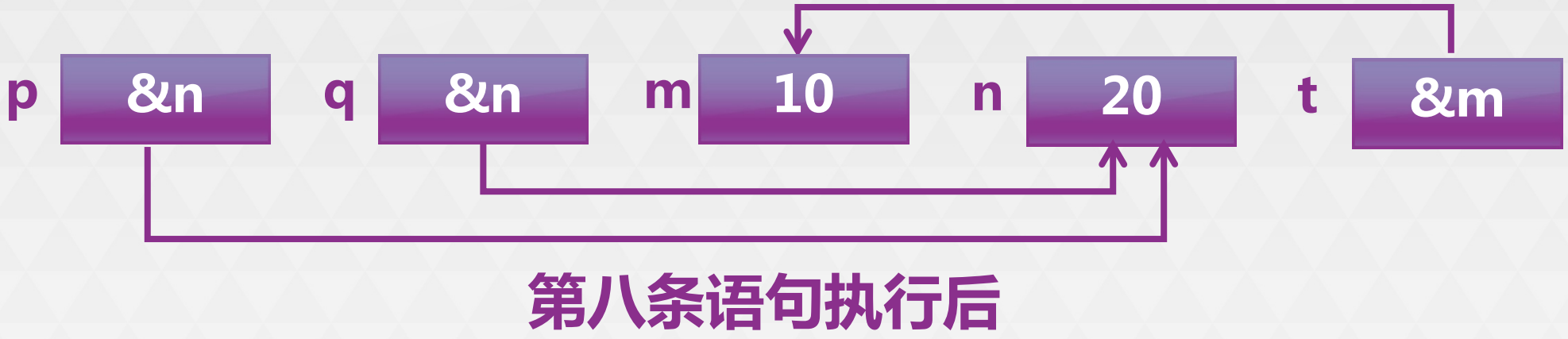
```
t = p;
```



第七条语句执行后

目标数据对象的操作

```
t = p;
p = q;
```

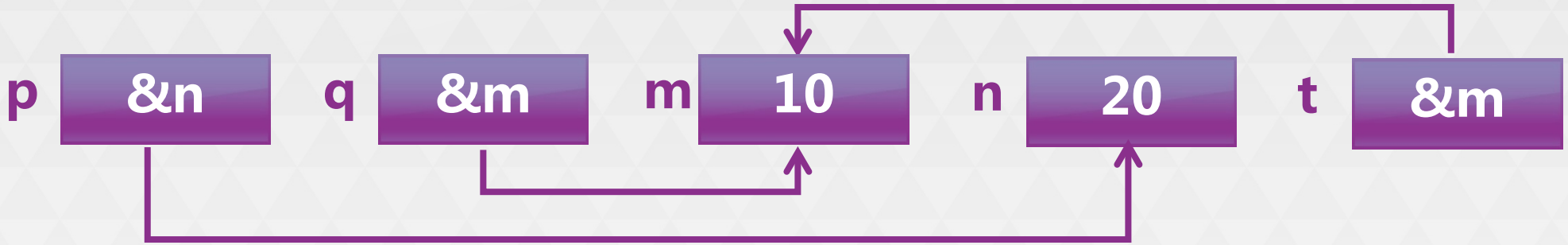


第八条语句执行后

目标数据对象的操作

```
t = p;
p = q;
q = t;
```

互换 p、q 的值，改变指针指向，m、n 的值没有变化



第九条语句执行后

指针的意义与作用

作为函数通信的一种手段

使用指针作为函数参数，不仅可以提高参数传递效率，还可以将该参数作为函数输出集的一员，带回结果

作为构造复杂数据结构的手段

使用指针构造数据对象之间的关联，形成复杂数据结构

作为动态内存分配和管理的手段

在程序执行期间动态构造数据对象之间的关联

作为执行特定程序代码的手段

使用指针指向特定代码段，执行未来才能实现的函数

指针与函数

数据交换函数

常量指针与指针常量

返回指针的函数

数据交换函数

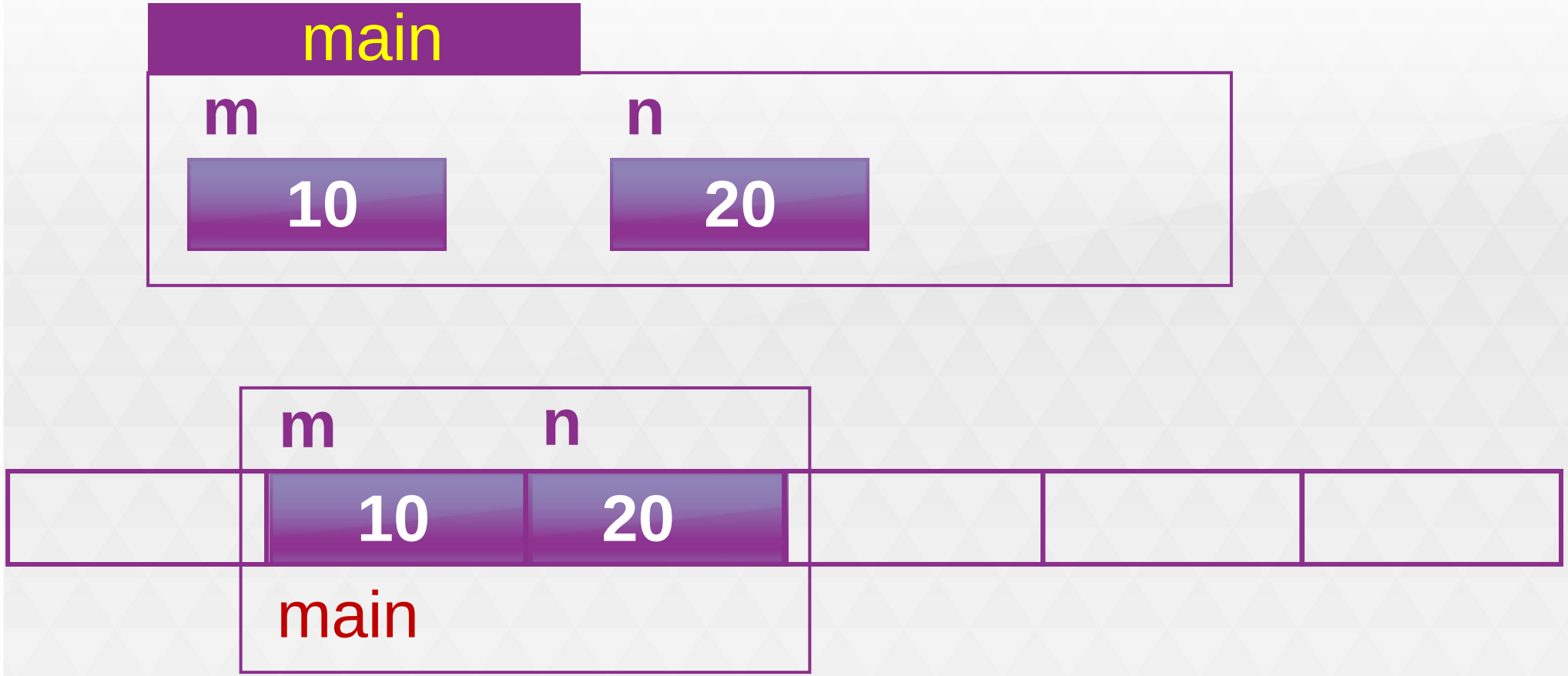
编写程序互换两个整型数据对象的值，要求使用函数实现数据对象值的互换

```
#include <iostream>
using namespace std;
void Swap( int * x, int * y );
int main()
{
  int m = 10, n = 20;
#ifndef NDEBUG
  cout << "main (before swapped): m = " << m << "; n = " << n << endl;
#endif
  Swap( &m, &n );  /* 调用Swap函数互换目标数据对象的值 */
#ifndef NDEBUG
  cout << "main (after swapped): m = " << m << "; n = " << n << endl;
#endif
  return 0;
}
```

数据交换函数

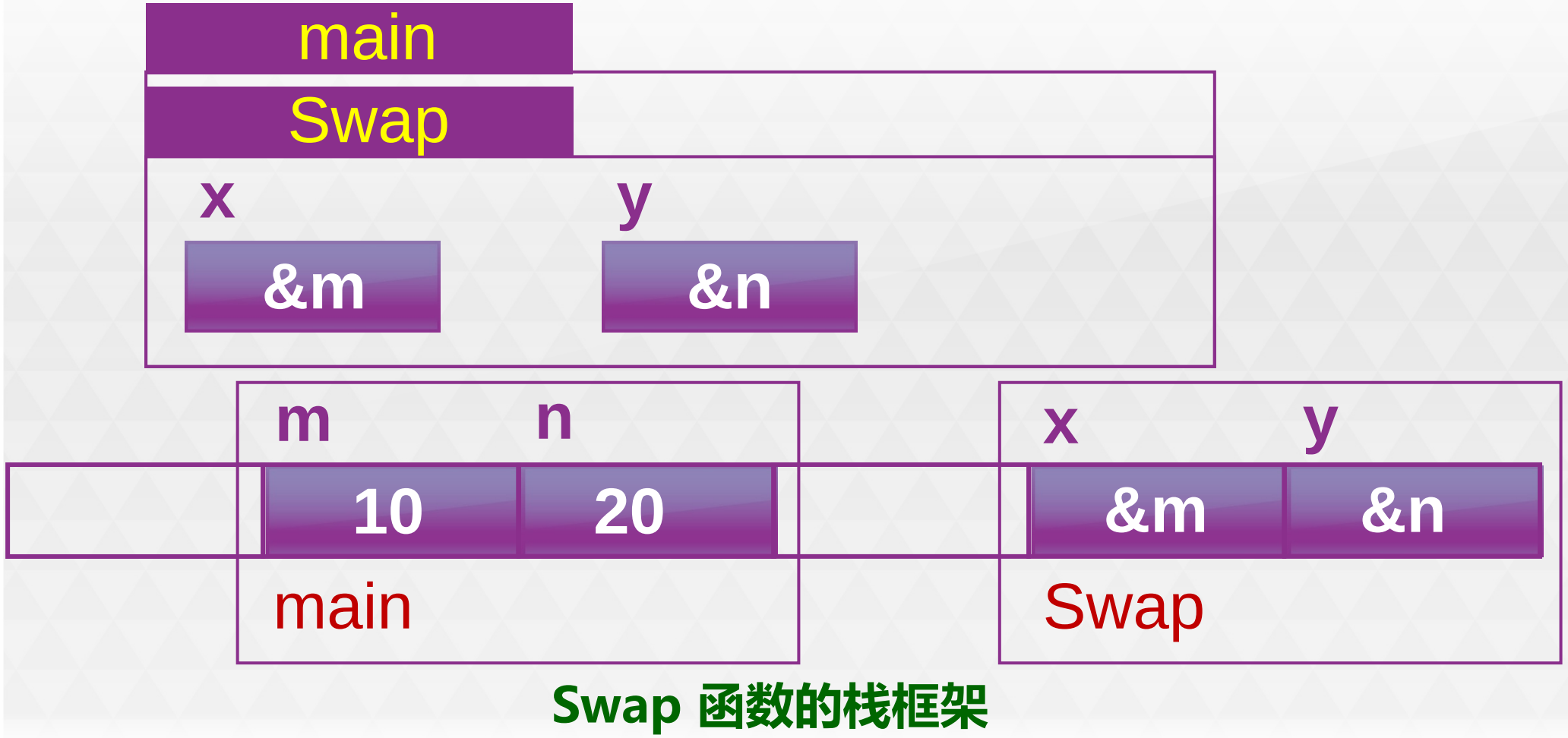
```
void Swap( int * x, int * y )
{
  int t;
  if( !x || !y ){
   cout << "Swap: Parameter(s) illegal." << endl;
   exit(1);
  }
#ifndef NDEBUG
  cout << "Swap (before swapped): *x = " << *x << "; *y = " << *y << endl;
#endif
  t = *x;
  *x = *y;
  *y = t;
#ifndef NDEBUG
  cout << "Swap (after swapped): *x = " << *x << "; *y = " << *y << endl;
#endif
}
```

函数调用栈框架



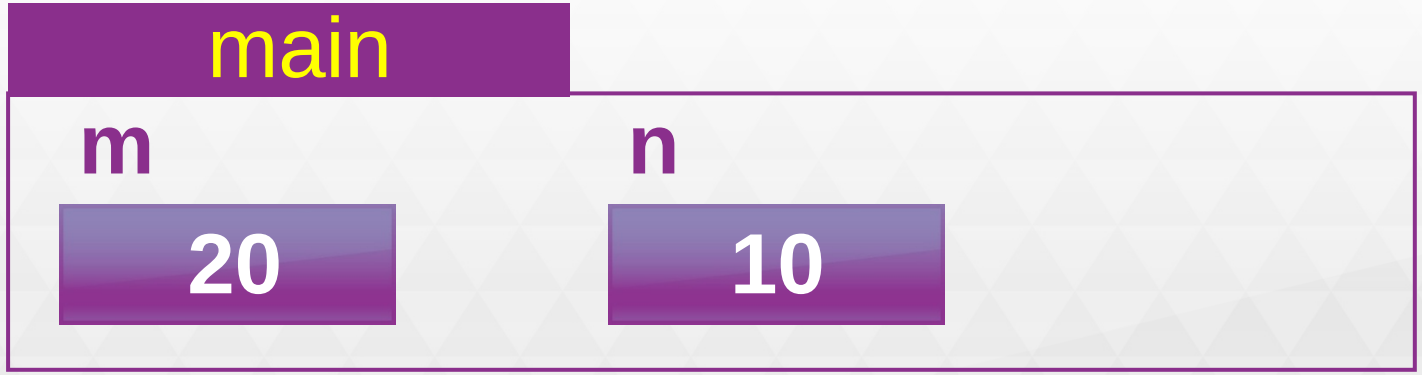
main 函数的栈框架（调用 Swap 函数前）

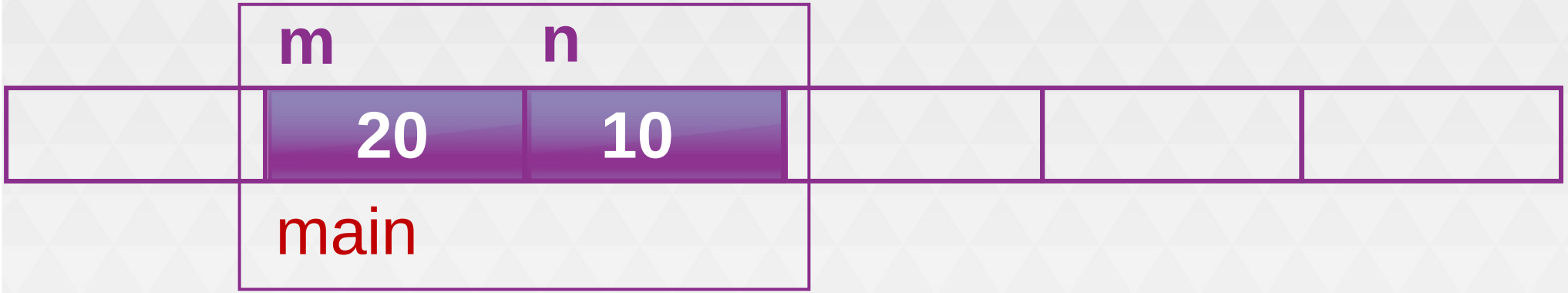
函数调用栈框架



Swap 函数的栈框架

函数调用栈框架





main 函数的栈框架（调用 Swap 函数后）

常量指针与指针常量

常量指针：指向常量的指针

性质：不能通过指针修改目标数据对象的值，但可以改变指针值，使其指向其他地方

示例一：int n = 10; const int * p = &n;

典型使用场合：作为函数参数，表示函数内部不能修改指针所指向的目标数据对象值

示例二：void PrintObject(const int * p);

指针常量：指针指向的位置不可变化

性质：不可将指针指向其他地方，但可改变指针所指向的目标数据对象值

示例三：int n = 10; int * const p = &n;

指针常量和其他常量一样，必须在定义时初始化

常量指针常量：指向常量的指针常量（指针的双重只读属性）

性质：指针值不可改变，指向的目标数据对象值也不可改变

示例四：const int n = 10; const int * const p = &n;

典型使用场合：主要作为函数参数使用

指针与函数返回值

指针类型可以作为函数返回值

- 函数内部返回某个数据对象的地址
- 调用函数后将返回值赋值给某个指针
- 特别说明：**不能返回函数内部定义的局部变量地址**

程序示例

```
int global = 0;
int * ReturnPointer()
{
  return &global;
}
```

指针与复合数据类型

指针与数组

- 数据对象地址的计算
- 作为函数参数的指针与数组
- 指针与数组的可互换性
- 多维数组参数的传递

指针与结构体

- 指向结构体的指针
- 指针作为结构体类型的成员

数据对象地址的计算

数组定义

```
int a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
```

数组基地址：&a 或 a

数组元素地址

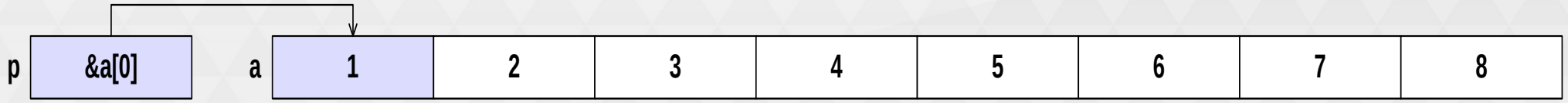
- **数组首元素地址：&a[0]**
- **数组第 i 元素地址：&a[0] + i * sizeof(int)**
- **数组基地址与首元素地址数值相同，故**
- **数组第 i 元素地址：a + i * sizeof(int)**

数据对象地址的计算

数组元素的地址

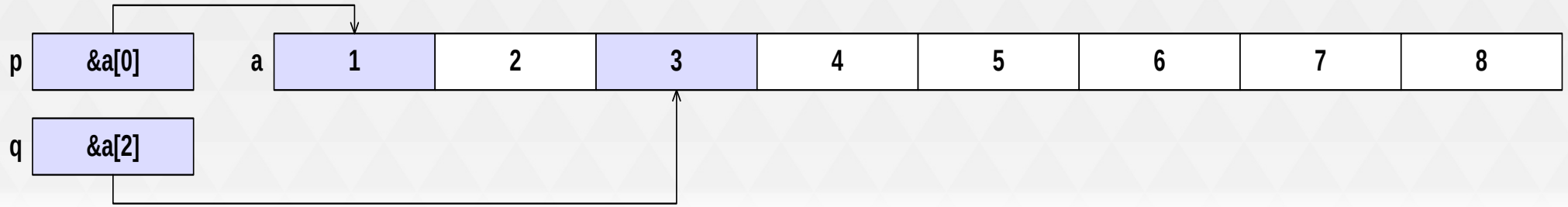
数组定义：int a[8] = {1, 2, 3, 4, 5, 6, 7, 8};

指针定义：int * p;　p = &a[0];　// p 指向数组首元素



指针定义：int * p;　p = a;　// p 仍指向数组首元素

指针定义：int * q;　q = &a[2];　// q 指向 a[2]



指针运算

希望表达 p、q 之间的联系

它们都指向同一数组中的元素

指针与整数加减运算

设 p 为指向整数数组中某元素的指针，i 为整数，则 p + i 表示指针向后滑动 i 个整数， p - i 表示指针向前滑动 i 个整数

例一：p 指向 a[0]，则 p + 2 指向 a[2]

例二：p 指向 a[3]，则 p - 2 指向 a[1]

指针与整数加减运算的结果仍为指针类型量，故可赋值

例三：p 指向 a[0]，则 q = p + 2 使得 q 指向 a[2]

指针运算

指针与整数加减运算规律

以指针指向的目标数据对象类型为单位，而不是以字节为单位

指针的递增递减运算

例四：p 指向 a[0]，则 p++ 指向 a[1]

例五：p 指向 a[1]，则 --p 指向 a[0]

指针减法运算

两个指针的减法运算结果为其间元素个数

例六： p 指向 a[0]， q 指向 a[2]，q - p 结果为 2

指针运算

指针关系运算

可以测试两个指针是否相等

例一：设 p、q 为指针，则 p == q 测试两个指针是否指向同一个目标数据对象

空指针：NULL

指针值 0：表示指针不指向任何地方，表示为 NULL

例二：设 p 为指针，则 p = NULL 表示 p 不指向任何目标数据对象

例三（测试指针 p 是否有意义）：if(p != NULL) 等价于 if(p)

使用指针前一定要测试其是否有意义！

作为函数参数的指针与数组

数组作为函数参数：函数定义

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

数组作为函数参数：函数调用

```
#define NUM_OF_ELEMENTS 8
int a[NUM_OF_ELEMENTS];
GenerateIntegers( a, NUM_OF_ELEMENTS );
```

作为函数参数的指针与数组

指针作为函数参数：函数定义

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

指针作为函数参数：函数调用

必须传递已分配空间的数组基地址

```
#define NUM_OF_ELEMENTS 8
int a[NUM_OF_ELEMENTS];
GenerateIntegers( a, NUM_OF_ELEMENTS );
```

数组元素的生成与逆序排列

随机生成8个整数保存到数组中，然后颠倒数组元素。要求使用指针。

```
//  "main.cpp"

#include <iostream>
using namespace std;

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

#define NUMBER_OF_ELEMENTS 8
```

数组元素的生成与逆序排列

```
//  "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 );
  ReverseIntegers( a, NUMBER_OF_ELEMENTS );
  cout << "After all elements of the array reversed: \n";
  PrintIntegers( a, NUMBER_OF_ELEMENTS );
  return 0;
}
```

数组元素的生成与逆序排列

```
//  "arrmanip.h"

void GenerateIntegers( int * p, unsigned int n );

void ReverseIntegers( int * p, unsigned int n );

void Swap( int * p, int * q );

void PrintIntegers( const int * p, unsigned int n );
```

数组元素的生成与逆序排列

```
//  "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;
```

数组元素的生成与逆序排列

```
//  "arrmanip.cpp"

void GenerateIntegers( int * p, unsigned int n )
{
  unsigned int i;
  Randomize();
  for( i = 0; i < n; i++ )
    *p++ = GenerateRandomNumber( lower_bound, upper_bound );
}
void ReverseIntegers( int * p, unsigned int n )
{
  unsigned int i;
  for( i = 0; i < n / 2; i++ )
    Swap( p + i, p + n - i - 1 );
}
```

数组元素的生成与逆序排列

```
//  "arrmanip.cpp"

void Swap( int * p, int * q )
{
  int t;    t = *p;    *p = *q;    *q = t;
}

void PrintIntegers( const int * p, unsigned int n )
{
  unsigned int i;
  for( i = 0; i < n; i++ )
    cout << setw(4) << *(p+i);
  cout << endl;
}
```

指针与数组的可互换性

互换情况

指针一旦指向数组的基地址，则使用指针和数组格式访问元素时的地址计算方式是相同的，此时可以互换指针与数组操作格式

程序示例

```
int a[3] = { 1, 2, 3 };    int * p = &a;    int i;
/* 正确，可以将指针 p 当作数组来处理  */
for( i = 0; i < 3; i++ )
  cout << p[i] << endl;
/* 正确，可以将数组 a 当作指针来处理  */
for( i = 0; i < 3; i++ )
  cout << *(a+i) << endl;
```

指针与数组的可互换性

例外情况

数组名为常数，不能在数组格式上进行指针运算

程序示例

```
/* 正确，指针 p 可赋值，指向下一元素 */
for( i = 0; i < 3; i++ )
  cout << *p++ << endl;

/* 错误，不能将数组 a 当作指针赋值 */
for( i = 0; i < 3; i++ )
  cout << *a++ << endl;
```

指针与数组的差异

使用指针或数组声明的数据对象性质不同

示例：int a[3] = { 1, 2, 3 };　int * p = &a;

定义数组的同时确定了数组元素的存储布局：a 为静态分配内存的数组；若 a 为全局数组，则程序执行前分配内存；若为局部数组，则在进入该块时分配内存

定义指针时规定指针数据对象的存储布局：p 为指针，若 p 为全局变量，则程序执行前分配内存；若为局部变量，则在进入该块时分配内存

定义指针时未规定目标数据对象的存储布局：p 为指针，指向一个已存在数组的基地址，即指向该位置处的整数 a[0]；若 p 未初始化，则目标数据对象未知

使用指针时，显式构造指针与目标对象的关联

多维数组作为函数参数

函数原型：正确例（有不妥，非错误）

直接传递元素个数也不妥当，只能处理固定元素个数的数组，应用场合十分受限

```
void PrintTwoDimensinalArray( int a[8][8], unsigned int m, unsigned int n );
```

函数原型：错误例

不能每维都不传递元素个数，语法规则不允许

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

函数原型：正确例（有不妥，非错误）

a 为指向数组基地址的整数指针，m 为第一维元素个数，n 为第二维元素个数，函数内部使用指针运算访问某个元素

```
void PrintTwoDimensinalArray( int * a, unsigned int m, unsigned int n );
```

如：第 i 行 j 列元素，使用指针运算 a + n * i + j 的结果指针指向

多维数组作为函数参数

```
//  函数定义
void PrintTwoDimensinalArray(int * a, unsigned int m, unsigned int n)
{
  unsigned int i, j;
  for( i = 0; i < m; i++ )
    for( j = 0; j < n; j++ )
      cout << *(a + n * i + j) << "; ";
}

//  函数调用
int a[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
PrintTwoDimensinalArray( a, 2, 3 );
```

指针与结构体

指向结构体的指针对象

```
struct STUDENT{ int id; STRING name; int age; };
STUDENT student = { 2007010367, "Name", 19 };
STUDENT * pstudent = &student;
```

访问指针所指向的结构体对象的成员

必须使用括号：选员操作符优先级高于引领操作符

```
(*pstudent).id = 2007010367;
(*pstudent).name = DuplicateString( "Name" );
(*pstudent).age = 19;
```

选员操作符“->”

```
pstudent->id = 2007010367;    // 不用书写括号，更方便
```

指针与结构体

结构体成员类型为指针

```
struct ARRAY{ unsigned int count; int * elements; };
int a[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
ARRAY array = { 8, &a };
```

访问指针类型的结构体成员

访问 elements 的第 i 个元素： array.elements[i]

若有定义：ARRAY * parray = &array;

访问 parray 指向的结构体对象 elements 的第 i 个元素：

(*parray).elements[i] 或 parray->elements[i]

结构体指针的使用场合

使用指向结构体对象的指针作为函数参数

好处一：节省结构体整体赋值的时间成本

好处二：解决普通函数参数不能直接带回结果的问题，可以在函数内部改变目标结构体对象的值

构造复杂的数据结构

动态创建和管理这些复杂的数据结构

动态数组：struct ARRAY { unsigned int count; int * elements; };

字符串

字符串的表示

三种理解角度：作为字符数组，作为指向字符的指针，作为抽象的字符串整体

字符数组与字符指针的差异

标准字符串库

字符串模板

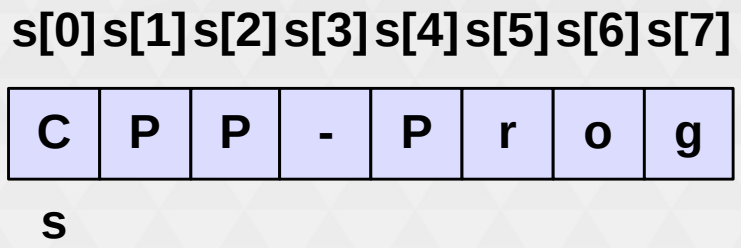
字符数组

字符数组的定义

与普通数组定义格式相同

示例：char s[8] = { 'C', 'P', 'P', '-', 'P', 'r', 'o', 'g' };

字符数组的存储布局



字符数组的访问

按照数组格式进行，逐一访问每个元素

很不方便

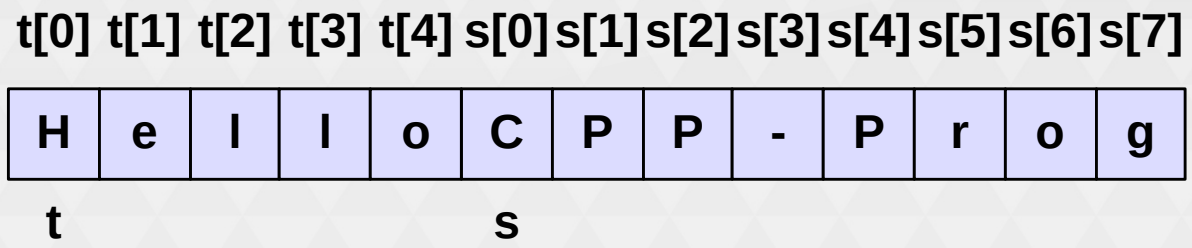
字符数组

多个字符数组连续存储时的问题

如何区分存储空间刚好连续的多个字符数组？

示例：char s[8] = { 'C', 'P', 'P', '-', 'P', 'r', 'o', 'g' };

char t[5] = { 'H', 'e', 'l', 'l', 'o' };



无法区分！

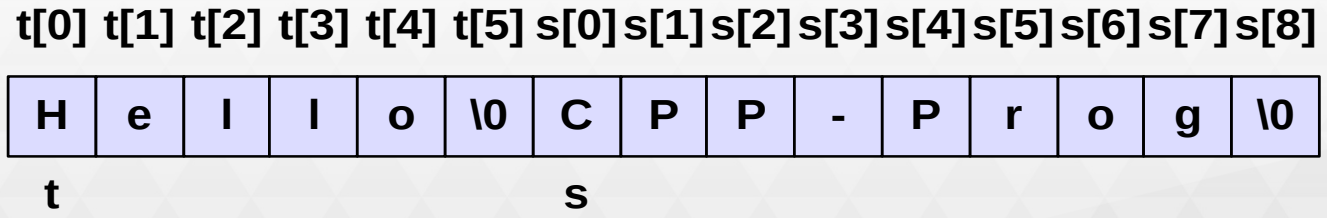
解决方案：字符数组末尾添加结束标志 ‘\0’

示例：char s[9] = { 'C', 'P', 'P', '-', 'P', 'r', 'o', 'g', '\0' };

char t[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };

字符数组

实际字符数组的存储布局



优　点

可以在程序运行时通过测试 ‘\0’ 字符确定字符数组是否结束，而不需要了解数组元素个数，使处理元素个数未知的数组成为可能

通过指针运算直接操作字符数组中的字符，而不再使用数组格式访问字符元素

特别说明

字符数组的分配空间必须能容纳额外的结束标志 ‘\0’

字符数组的访问

编写函数，返回字符 c 在字符串 s 中的首次出现位置

```
unsigned int FindCharFirst( char c, char s[] )
{
  unsigned int i;
  if( !s )
  {
    cout << "FindCharFirst: Illegal string.\n";
    exit(1);
  }
  for( i = 0; s[i] != '\0'; i++ )
  {
    if( s[i] == c )
      return i;
  }
  return inexistent_index;   // 0xFFFFFFFF
}
```

字符指针

编写函数，返回字符 c 在字符串 s 中的首次出现位置

```
unsigned int FindCharFirst( char c, char * s )
{
  char * t;
  if( !s )
  {
    cout << "FindCharFirst: Illegal string.\n";
    exit(1);
  }
  for( t = s; *t != '\0'; t++ )
  {
    if( *t == c )
      return t - s;
  }
  return inexistent_index;   // 0xFFFFFFFF
}
```

抽象字符串

抽象字符串的定义

```
typedef char * STRING;
typedef const char * CSTRING;
```

FindCharFirst 函数的声明格式

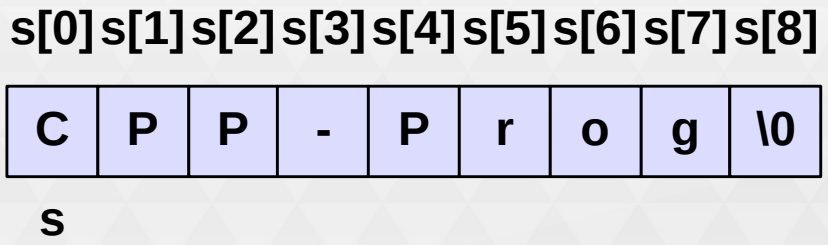
下述三种声明格式完全相同：

```
unsigned int FindCharFirst( char c, char s[] );
unsigned int FindCharFirst( char c, char* s );
unsigned int FindCharFirst( char c, STRING s );
```

字符数组与字符指针的差异

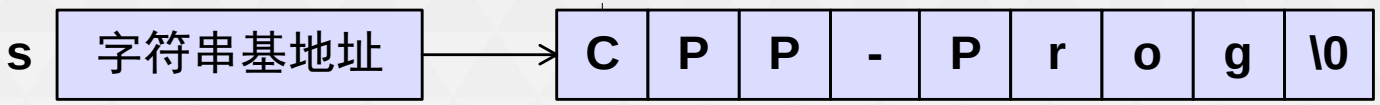
字符数组量的定义、初始化与存储

```
char s[9] = { 'C', 'P', 'P', '-', 'P', 'r', 'o', 'g', '\0' };
```



字符指针量的定义、初始化与存储

```
char * s = "CPP-Prog";
```



后者存在单独的指针变量，前者则没有

字符数组与字符指针的差异

按指针格式定义字符串，可以直接赋值

示例：char * s;　s = "CPP-Prog";　// 正确

字符串文字首先分配空间，然后将其基地址赋给 s，使 s 指向该字符串基地址

按字符数组格式定义字符串，不能直接赋值

示例：char s[9];　s = "CPP-Prog";　// 错误

不能对数组进行整体赋值操作

原因：数组空间已分配，字符串文字空间已分配，且它们位于不同位置，不能直接整体复制

返回字符串的函数：正确例

编写函数，将某个字符 c 转换为字符串

```
STRING TransformCharIntoString( char c )
{
  STRING _s = (STRING)malloc( 2 );
  _s[0] = c;
  _s[1] = '\0';
  return _s;
}
```

返回字符串的函数：错误例

编写函数，将某个字符 c 转换为字符串

```
char * TransformCharIntoString( char c )
{
  char _s[2];
  _s[0] = c;
  _s[1] = '\0';
  return _s;
}
```

错误函数定义

对于所有返回值为指针类型的函数，都不能返回在函数内部定义的局部数据对象——所有局部对象在函数结束后不再有效，其地址在函数返回后没有意义

标准字符串库

标准库中关于字符串处理的函数很多，均定义于头文件“cstring”中

常用字符串函数

```
char * strcat( char * dest, const char * src );
int strcmp( const char * s1, const char * s2 );
char * strcpy( char * dest, const char * src );
int strlen( const char * s );
char * strtok( char * token, const char * delimiters );
```

不建议使用标准字符串库，使用 string 类替代

string类

定义于头文件“string”中

声明与构造string对象

```
string s = "abcdefg";
string s( "abcdefg" );
```

读取与写入string对象

```
cout << s << endl;
cin >> s;  // 读取以空格、制表符与回车符分隔的单词
getline( cin, s, '\n' );  // 读取包含空格和制表符在内的整行
```

string类

获取string对象的长度

```
string s = "abcdefg";
int a = s.length();
```

改变string对象的容量大小

```
s.resize(32); // 将s设为32字符长，多余舍弃，不足空闲
s.resize(32, '='); // 多余舍弃，不足补‘=’
```

string对象的追加操作

```
string s1 = "abcd", s2 = "efg";
s1.append( s2 ); // 将字符串s2追加到s1尾部
```

string类

string对象的比较操作

```
string s1 = "abcdefg", s2 = "abcdxyz";
int a = s1.compare( s2, 0 );  // 从0号位字符开始比较
```

string对象的查找操作

```
string s1 = "abcdefg", s2 = "bcd";
int a = s1.find( s2, 0 );  // 从字符串开头开始查找，结果为
s2在s1中首次出现的位置
```

动态存储管理

内存分配

标准库的动态存储管理函数

- 内存分配函数 malloc
- 内存释放函数 free
- C++ 的内存分配操作符：new 与 delete

关于动态存储管理的若干注意事项的说明

动态数组

内存分配与释放

静态内存分配方式

适用对象：全局变量与静态局部变量

分配与释放时机：在程序运行前分配，程序结束时释放

自动内存分配方式

适用对象：普通局部变量

分配与释放时机：在程序进入该函数或该块时自动进行，退出时自动释放

动态内存分配方式

适用对象：匿名数据对象（指针指向的目标数据对象）

分配与释放时机：在执行特定代码段时按照该代码段的要求动态分配和释放

动态内存分配的性质

动态内存分配的目的

静态与自动内存分配方式必须事先了解数据对象的格式和存储空间大小

部分场合无法确定数据对象的大小

示例：声明一个包含 n 个元素的整数数组，n 值由用户在程序执行时输入。

编译时程序未执行，不知道 n 值！

动态内存分配的位置

计算机维护的一个专门存储区：堆

所有动态分配的内存都位于堆中

动态内存分配的关键技术

使用指针指向动态分配的内存区

使用引领操作符操作目标数据对象

标准库的动态存储管理函数

动态存储管理函数的原型

头文件：“cstdlib”和“cmalloc”，两者包含其一即可

内存分配函数原型：void * malloc(unsigned int size);

内存释放函数原型：void free(void * memblock);

void * 类型

特殊的指针类型，指向的目标数据对象类型未知

不能在其上使用引领操作符访问目标数据对象

可以转换为任意指针类型，不过转换后类型是否有意义要看程序逻辑

可以在转换后的类型上使用引领操作符

主要目的：作为一种通用指针类型，首先构造指针对象与目标数据对象的一般性关联，然后由程序员在未来明确该关联的性质

malloc 函数

malloc 函数的一般用法

首先定义特定类型的指针变量：char * p;

调用 malloc 函数分配内存：p = (char *)malloc(11);

参数表示所需要分配的存储空间大小，以字节为单位

例：若要分配能够保存 10 个字符的字符串，分配 11 个字节（字符串结束标志也要分配空间）

将返回值转换为 char * 类型赋值给原指针，使 p 指向新分配空间的匿名目标数据对象

malloc 函数示例

编写函数，复制字符串

```
char * DuplicateString( char * s )
{
  char * t;
  unsigned int n, i;
  if( !s ){ cout << "DuplicateString: Parameter Illegal.";  exit(1);  }
  n = strlen( s );
  t = ( char * )malloc( n + 1 );
  for( i = 0; i < n; i++ )
    t[i] = s[i];
  t[n] = '\0';
  return t;
}
```

free 函数

free 函数的一般用法

传递一个指向动态分配内存的目标数据对象的指针

示例一：char * p;　p = (char *)malloc(11);　free(p);

示例二：int * p = (int *)malloc(10 * sizeof(int));　free(p);

示例二分配能够容纳 10 个整数的连续存储空间，使 p 指向该空间的基地址，最后调用 free 函数释放 p 指向的整个空间

特别说明：有分配就有释放

free 函数释放的是 p 指向的目标数据对象的空间，而不是 p 本身的存储空间

调用 free 函数后，p 指向的空间不再有效，但 p 仍指向它

为保证在释放目标数据对象空间后，不会再次使用 p 访问，建议按照下述格式书写代码：free(p);　p = NULL;

new/new[ ] 操作符

动态创建单个目标数据对象

分配目标对象：int * p; p = new int; *p = 10;

分配目标对象：int * p; p = new(int); *p = 10;

分配目标对象并初始化：int * p; p = new int(10); // 将 *p 初始化为 10

分配目标对象并初始化：int * p; p = new(int)(10);

动态创建多个目标数据对象

分配数组目标对象：int * p; p = new int[8]; // 分配 8 个元素的整数数组

delete/delete[ ] 操作符

释放单个目标数据对象

释放目标对象：int * p; p = new int; *p = 10; delete p;

释放多个目标数据对象

释放数组目标对象：int * p; p = new int[8]; delete[] p;

不是delete p[ ]!

所有权与空悬指针

目标数据对象的所有权

指向该目标数据对象的指针对象拥有所有权

在程序中要时刻明确动态分配内存的目标数据对象的所有权归属于哪个指针数据对象

指针使用的一般原则

主动释放原则：如果某函数动态分配了内存，在函数退出时该目标数据对象不再需要，应主动释放它，此时 malloc 与 free 在函数中成对出现

所有权转移原则：如果某函数动态分配了内存，在函数退出后该目标数据对象仍然需要，此时应将其所有权转交给本函数之外的同型指针对象，函数内部代码只有 malloc，没有 free

所有权与空悬指针

空悬指针问题

所有权的重叠：指针赋值操作导致两个指针数据对象指向同样的目标数据对象，即两个指针都声称“自己拥有目标数据对象的所有权”

示例：int *p, *q;　q = (int*)malloc(sizeof(int));　p = q;

产生原因：如果在程序中通过某个指针释放了目标数据对象，另一指针并不了解这种情况，它仍指向不再有效的目标数据对象，导致空悬指针

示例：free(p);　p = NULL; // q 为空悬指针，仍指向原处

所有权与空悬指针

解决方案

确保程序中只有惟一一个指针拥有目标数据对象，即只有它负责目标数据对象的存储管理，其它指针只可访问，不可管理；若目标数据对象仍有存在价值，但该指针不再有效，此时应进行所有权移交

在一个函数中，确保最多只有一个指针拥有目标数据对象，其它指针即使存在，也仅能访问，不可管理

如果可能，在分配目标数据对象动态内存的函数中释放内存，如 main 函数分配的内存在 main 函数中释放

退一步，如果上述条件不满足，在分配目标数据对象动态内存的函数的主调函数中释放内存，即将所有权移交给上级函数

级级上报，层层审批

内存泄露与垃圾回收

内存泄露问题

产生原因：若某个函数通过局部指针变量动态分配了一个目标数据对象内存，在函数调用结束后没有释放该内存，并且所有权没有上交

示例：`void f(){ int * p = new int; *p = 10; }`

函数 f 结束后，p 不再存在，*p 所在的存储空间仍在，10 仍在，但没有任何指针对象拥有它，故不可访问

问题的实质：**动态分配的内存必须动态释放，函数本身并不负责管理它**

垃圾回收机制：系统负责管理，程序员不需要主动释放动态分配的内存，Java 有此功能，C 语言无

垃圾回收机制在需要时效率很差，而不需要时效率很好

引用类型

引用的定义

定义格式：数据类型& 变量名称 = 被引用变量名称；

示例：int a; int & ref = a;

引用的性质

引用类型的变量不占用单独的存储空间

为另一数据对象起个别名，与该对象同享存储空间

特殊说明

引用类型的变量必须在定义时初始化

此关联关系在引用类型变量的整个存续期都保持不变

对引用类型变量的操作就是对被引用变量的操作

引用示例

编写程序，使用引用操作目标数据

```
#include <iostream>
using namespace std;
int main(){
  int a;
  int & ref = a;
  a = 5;
  cout << "a: " << a << endl;  cout << "ref: " << ref << endl;
  ref = 8;
  cout << "a: " << a << endl;  cout << "ref: " << ref << endl;
  return 0;
}
```

引用作为函数参数

引用的最大意义：作为函数参数

参数传递机制：引用传递，直接修改实际参数值

使用格式：返回值类型 函数名称(类型 & 参数名称)；

函数原型示例：void Swap(int & x, int & y);

函数实现示例：

```
void Swap( int & x, int & y ){
   int t;  t = x;  x = y;  y = t;  return;
}
```

函数调用示例：

```
int main(){
   int a = 10, b = 20;  Swap( a, b );  return 0;
}
```

引用作为函数返回值

常量引用：仅能引用常量，不能通过引用改变目标对象值；引用本身也不能改变引用对象

引用作为函数返回值时不生成副本

函数原型示例：int & Inc(int & dest, const int & alpha);

函数实现示例：

```
int & Inc( int & dest, const int & alpha ){
    dest += alpha;  return dest;
}
```

函数调用示例：引用类型返回值可以递增

```
int main(){
    int a = 10, b = 20, c;  Inc( a, b );  c = Inc(a, b)++;  return 0;
}
```

编程实践

7.1 设计并实现有理数库。使用整数表示有理数的分子与分母，完成有理数的加减乘除与化简运算。

7.2 继续编程实践题6.3。完成桥牌库的初步设计与实现。