CS 1502H 程序设计思想与方法 (C++)

Chapter 7. 指针-1

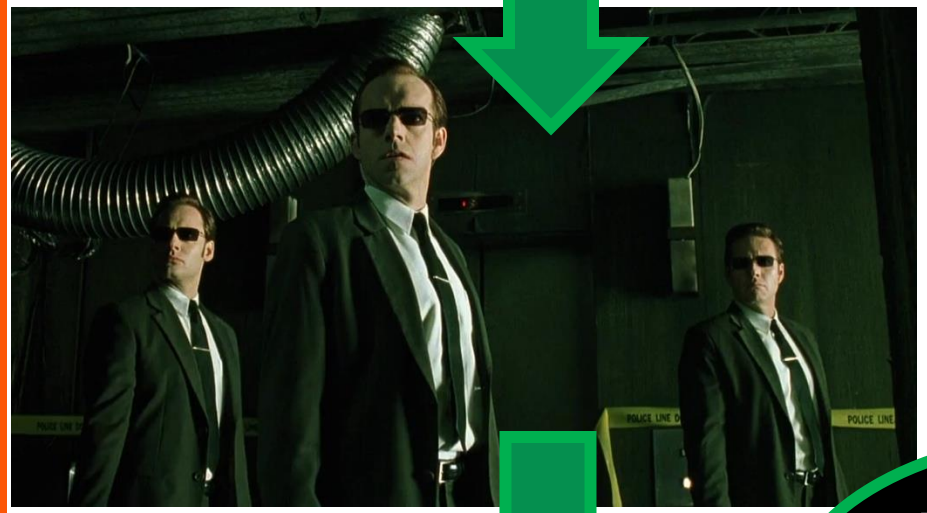
陈东尧
上海交通大学
2025年秋季

本讲大纲

- **指针概念及指针变量的定义**
- **指针运算**
- **指针与数组**
- **动态内存分配**
- **字符串与指针**

为什么要学指针？

高级语言中以变量名来访问数据，但机器码使用的是内存地址。



了解运行机制有助于高效解决问题

Credit: The Matrix, Warner Bros

为什么要学指针？

- 例子：【值传递】是否还记得swap函数？
  - 如何成功交换x和y的值？

```
void swap(int x, int y);
{
    int temp;
    temp=x;
    x=y;
    y=temp;
}
```

为什么要学指针？

- 例子：【值传递】是否还记得swap函数？
- 学会指针可以对计算机程序更全面掌握

为什么要学指针？

- 例子：【值传递】是否还记得swap函数？
- 学会指针可以对计算机程序更全面掌握

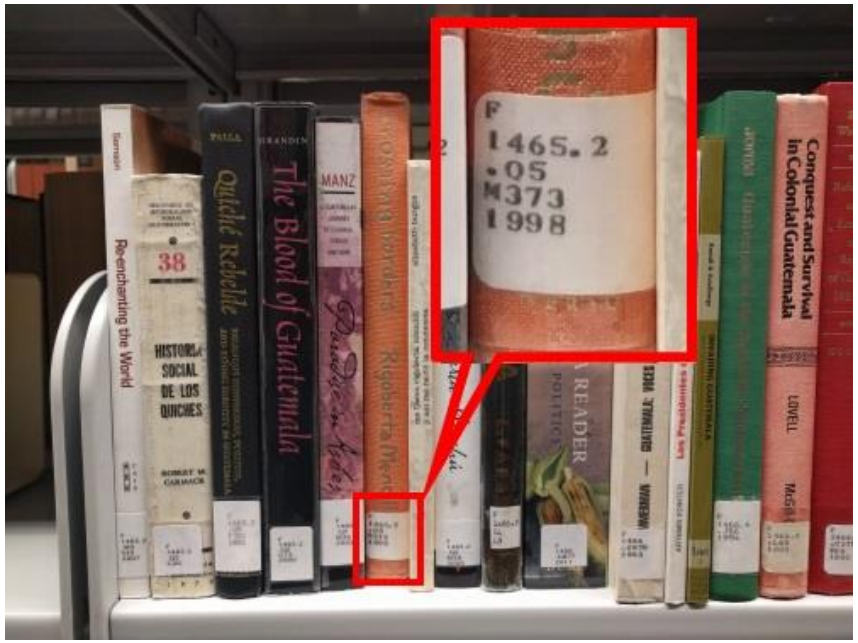
为什么要学指针？

- 例子：【值传递】是否还记得swap函数？
- 学会指针可以对计算机程序更全面掌握
  - 更深层次的程序运行逻辑
  - 代码加速
  - …

指针的概念，要了解此先要理解什么是引用

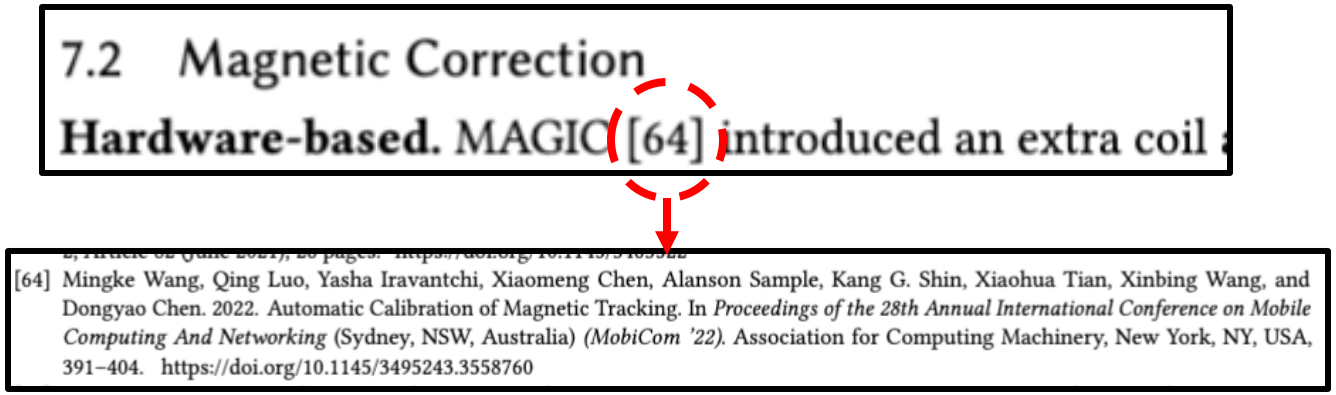
- 引用（reference）

如何找书?



指针的概念，要了解此先要理解什么是引用

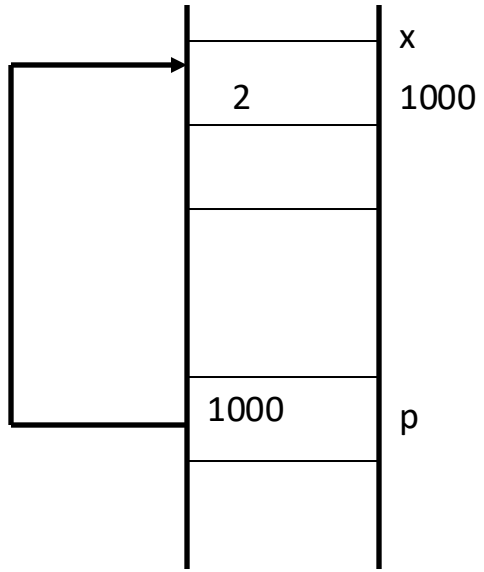
- 引用（reference）



如何整理参考文献?

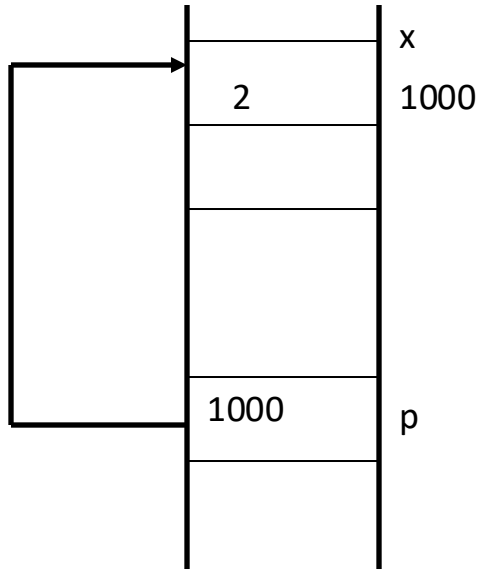
指针的概念

- 指针
  - 地址**作为数据**处理



指针的概念

- 指针
  - 地址作为数据处理

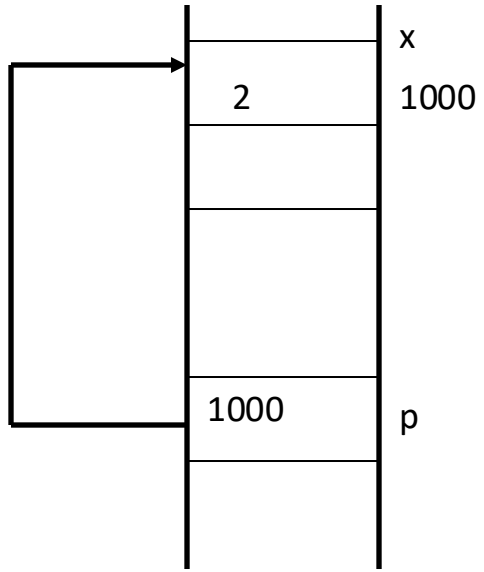



```
#include <iostream>
using namespace std;
int main (){
        int var1;
        char var2[10];
        cout << "var1 变量的地址:  ";
        cout << &var1 << endl;
        cout << "var2 变量的地址:  ";
        cout << &var2 << endl;
        return 0;
}
```

```
var1 变量的地址:  0x7ff7b9f4dfc4
var2 变量的地址:  0x7ff7b9f4dfce
```

指针的概念

- 指针
  - 地址作为数据处理



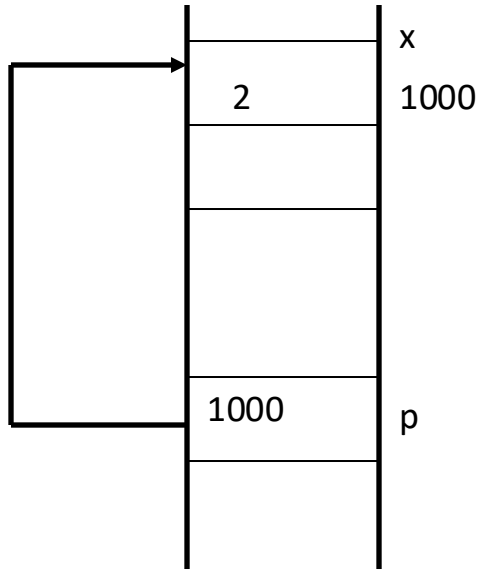
& **是什么?**

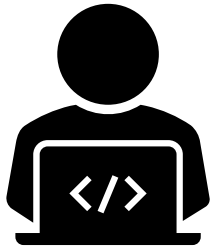
```
#include <iostream>
using namespace std;
int main (){
        int var1;
        char var2[10];
        cout << "var1 变量的地址:  ";
        cout << &var1 << endl;
        cout << "var2 变量的地址:  ";
        cout << &var2 << endl;
        return 0;
}
```

```
var1 变量的地址:  0x7ff7b9f4dfc4
var2 变量的地址:  0x7ff7b9f4dfce
```

指针的概念

- 指针
  - 地址作为数据处理
- 用途
  - 提供间接访问
  - 实现动态内存分配





代码演示

chapter7-pointer-trivia.cpp

```cpp
#include <iostream>
using namespace std;
int main (){
        int var1;
        char var2[10];
        cout << "var1 变量的地址:  ";
        cout << &var1 << endl;
        cout << "var2 变量的地址:  ";
        cout << &var2 << endl;
        return 0;
}
```

```
var1 变量的地址:  0x7ff7b9f4dfc4
var2 变量的地址:  0x7ff7b9f4dfce
```

指针变量定义

- 指针变量定义
  - 该变量中存放的是一个地址
  - 指针变量的主要用途是提供间接访问，因此也需要知道指针指向的单元的数据类型

- 指针变量的**定义格式**：type *variable；（*符号，asterisk mark）
  - 如：int *intptr;
  - double *doubleptr;
  - int *p，x，*q;

https://www.runoob.com/cplusplus/cpp-pointers.html

指针变量定义

- 指针变量定义
  - 该变量中存放的是一个地址
  - 指针变量的主要用途是提供间接访问，因此也需要知道指针指向的单元的数据类型

- 指针变量的定义格式：type *variable；（*符号，asterisk mark）
  - 如：int *intptr;
  - double *doubleptr;
  - int *p, x, *q;

- 统配指针 void *
  - 可以指向任何类型

```
int a = 10;
char b = 'x';

void *p = &a;  // void pointer holds address of int 'a'
p = &b; // void pointer holds address of char 'b'
```

https://www.runoob.com/cplusplus/cpp-pointers.html

空指针 （null pointer）

- 不指向任何地址， 空指针可表示为
  - 用符号常量NULL， NULL表示值为0的int;
  - 用符号常量nullptr（ C++11的风格）， 表示地址为零的指针

```
int* x = nullptr;
```

- NULL的值是整数0， 与函数重载配合时可能会出问题。
  - 如有两函数:

    void f(int *);

    void f(int);

    f(NULL)会调用f(int)

    nullptr **不存在到整型的隐式转换**

本讲大纲

- **指针概念及指针变量的定义**
- **指针运算**
- **指针与数组**
- **动态内存分配**
- **字符串与指针**

指针变量的操作

- **赋值: 让指针指向某一变量**
  - **取地址运算符** “&”（ampersand mark）：如表达式 “&x” 返回的是变量 x 的地址。如：intptr = &x;
  - **& 运算符后面不能跟常量或表达式**。如 &2 是没有意义的，&(m * n + p)。也是没有意义的

指针变量的操作

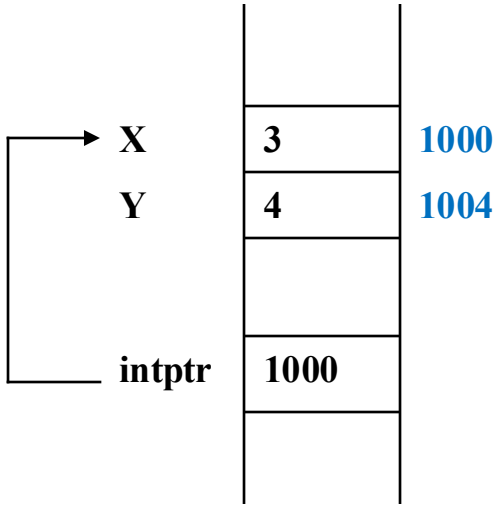
- **赋值: 让指针指向某一变量**
  - **取地址（reference）运算符** “&”（ampersand mark）： 如表达式 “&x” 返回的是变量 x 的地址。如：intptr = &x;
  - **& 运算符后面不能跟常量或表达式**。如 &2 是没有意义的，&(m * n + p)。也是没有意义的

- **间接访问指针的元素**
  - **解引用（dereference）运算符** “*” （ asterisk mark ）： *intptr表示的是 intptr 指向的这个单元。如：*intp = 5 等价于 x = 5
  - **在对 intptr 使用引用运算之前，必须先对 intptr 赋值**

指针实例操作

如有：

```
int X, *intptr, Y;
X=3;
Y=4;
intptr=&X;
```



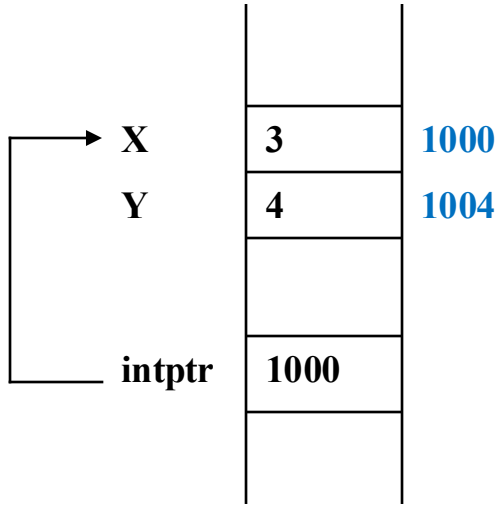
如执行：

```
*intptr=Y+4;
```

指针实例操作

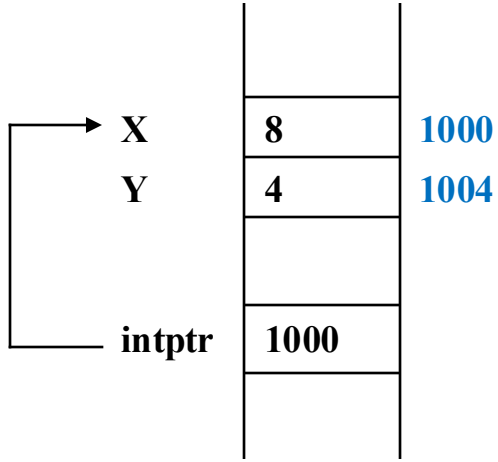
如有：

```
int X, *intptr, Y;
X=3;
Y=4;
intptr=&X;
```



如执行：

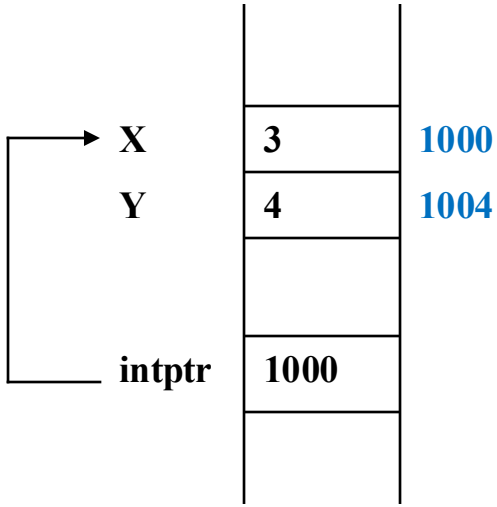
```
*intptr=Y+4;
```



指针实例操作

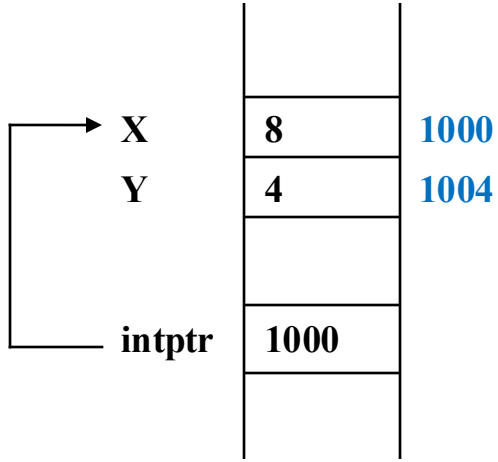
如有：

```
int X, *intptr, Y;
X=3;
Y=4;
intptr=&X;
```



如执行：

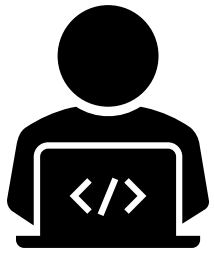
```
*intptr=Y+4;
```



注意：不能用　`intptr=100;`

因为我们不知道存储变量的真实地址

指针变量操作例子

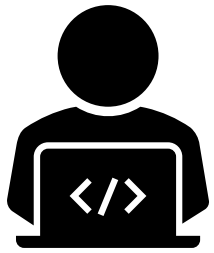


代码演示

chapter7-pointer-operation-easy.cpp

```cpp
#include <iostream>
using namespace std;
int main ()
{
    int var = 20; // 实际变量的声明
    int *iptr; // 指针变量的声明
    iptr = &var; // 在指针变量中存储 var 的地址
    cout << "Value of var variable: ";
    cout << var << endl;
    // 输出在指针变量中存储的地址
    cout << "Address stored in ip variable: ";

    // 访问指针中地址的值
    cout << "Value of *ip variable: ";

    return 0;
}
```

指针变量操作例子

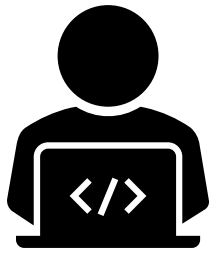


代码演示

chapter7-pointer-operation-easy.cpp

```cpp
#include <iostream>
using namespace std;
int main ()
{
    int var = 20; // 实际变量的声明
    int *iptr; // 指针变量的声明
    iptr = &var; // 在指针变量中存储 var 的地址
    cout << "Value of var variable: ";
    cout << var << endl;
    // 输出在指针变量中存储的地址
    cout << "Address stored in ip variable: ";
    cout << iptr << endl;
    // 访问指针中地址的值
    cout << "Value of *ip variable: ";

    return 0;
}
```

指针变量操作例子



代码演示

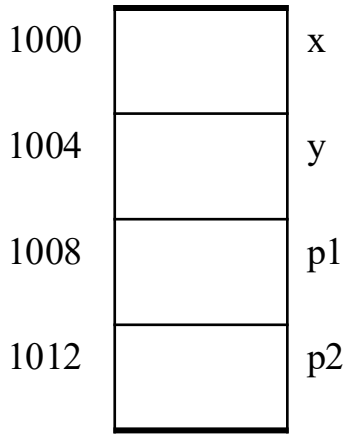
chapter7-pointer-operation-easy.cpp

```cpp
#include <iostream>
using namespace std;
int main ()
{
    int var = 20; // 实际变量的声明
    int *iptr; // 指针变量的声明
    iptr = &var; // 在指针变量中存储 var 的地址
    cout << "Value of var variable: ";
    cout << var << endl;
    // 输出在指针变量中存储的地址
    cout << "Address stored in ip variable: ";
    cout << iptr << endl;
    // 访问指针中地址的值
    cout << "Value of *ip variable: ";
    cout << * iptr << endl;
    return 0;
}
```

指针变量的使用

设有定义

```
int x, y;
int *p1,*p2;
```



执行语句：

```
x=23;
y=234;
```

执行语句：

```
p1=&x;
p2=&y;
```

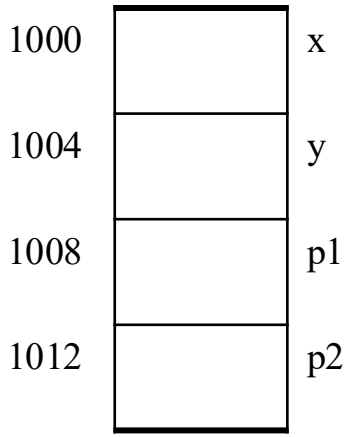
执行语句：

```
*p1=34;
p2=p1;
```

指针变量的使用

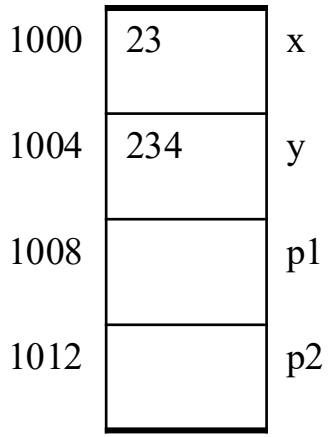
设有定义

```
int x, y;
int *p1,*p2;
```



执行语句：

```
x=23;
y=234;
```



执行语句：

```
p1=&x;
p2=&y;
```

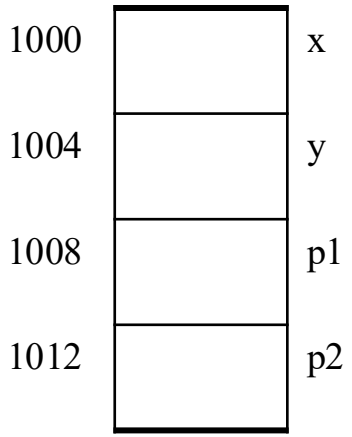
执行语句：

```
*p1=34;
p2=p1;
```

指针变量的使用

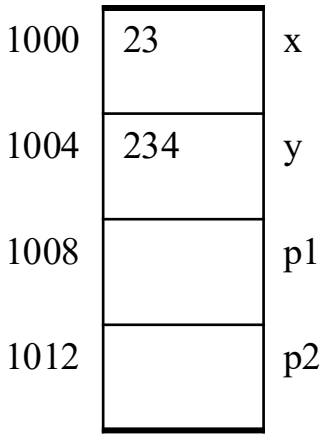
设有定义

```
int x, y;
int *p1,*p2;
```



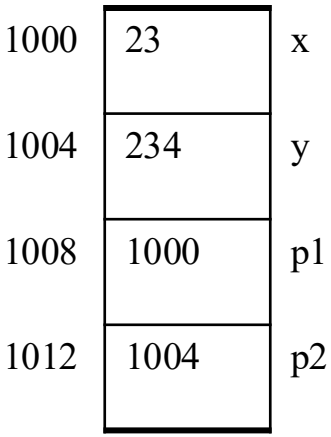
执行语句：

```
x=23;
y=234;
```



执行语句：

```
p1=&x;
p2=&y;
```



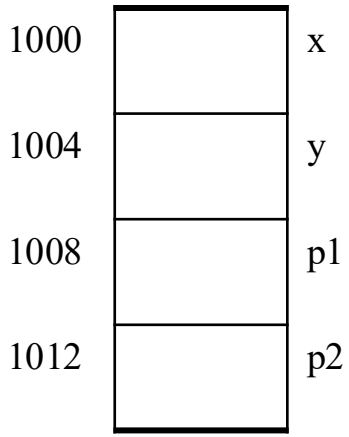
执行语句：

```
*p1=34;
p2=p1;
```

指针变量的使用

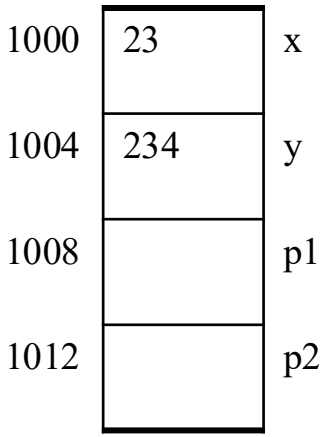
设有定义

```
int x, y;
int *p1,*p2;
```



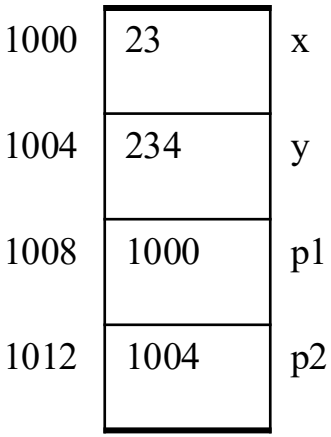
执行语句：

```
x=23;
y=234;
```



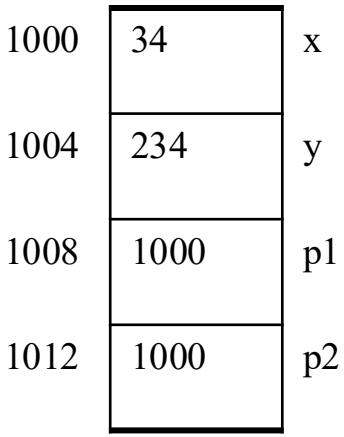
执行语句：

```
p1=&x;
p2=&y;
```



执行语句：

```
*p1=34;
p2=p1;
```



类型转换：reinterpret_cast

- 指针只能指向同类变量
  - int x, *iptr = &x;   // 正确
  - int x; float *fptr = &x;   // 错误

类型转换：reinterpret_cast

- 指针只能指向同类变量
  - int x, *iptr = &x;   // 正确
  - int x; float *fptr = &x;   // 错误
- 作用
  - 重新解释内存中的二进制比特串
  - 如将 int 的 32 位二进制数解释成 float 类型的指针
  - float *fp = reinterpret_cast < float * > (&x);   //正确

本讲大纲

- **指针概念及指针变量的定义**
- **指针运算**
- **指针与数组**
- **动态内存分配**
- **字符串与指针**

指向数组元素的指针

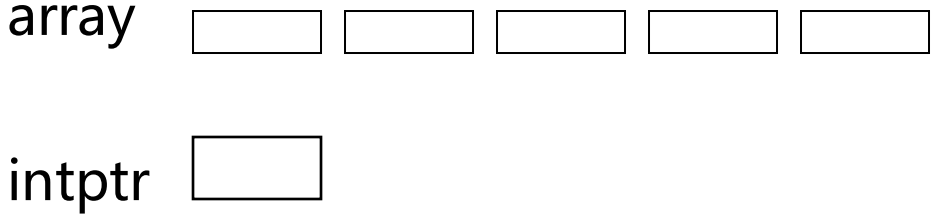
- **指向数组元素的指针**
  - p = &a[1], p = &a[i]
- **数组元素的地址是通过数组首地址计算的**
  - 如数组的首地址是 1000，则第 i 个元素的地址是 1000 + i * 每个数组元素所占的空间长度
- **数组名是指针常量**
  - 如有 int array[10], *p;
  - 可以执行 p = array;

指针和数组

- **一旦执行了p=array，对该指针可以进行任何有关数组下标的操作**
- 访问array的10个元素，可用下列循环

```
for（p=array, i=0; i<10; ++i）
    cout << p[i];
```

- 注意：数组和指针是不同的：

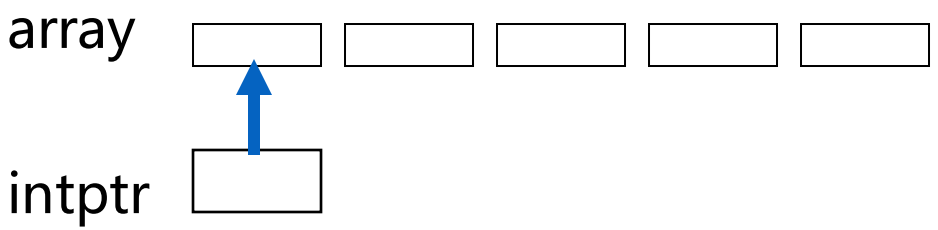


指针和数组

- **一旦执行了p=array，对该指针可以进行任何有关数组下标的操作**
- 访问array的10个元素，可用下列循环

```
for（p=array, i=0; i<10; ++i）
    cout << p[i];
```

- 注意：数组和指针是不同的：



当执行了
intptr = array
后

指针的算数运算

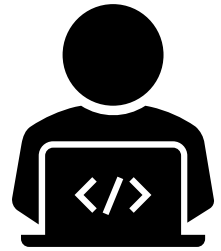
指针保存的是一个地址，地址是一个整型数，因此可以进行各种算术运算

- 指针只能做加减运算
- 指针加减是**加减一个基类型**的长度
- 仅当指针指向数组时，加减运算才有意义

指针运算的意义

- 指针+1表示数组中指针指向元素的下一元素地址；
- 指针-1表示数组中指针指向元素的上一元素地址；
- 合法的指针操作：p + k, p - k, p1 - p2

举例



代码演示

chapter7-pointer-cast.cpp

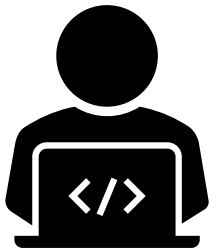
```
int array[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *p;

cout<<&p<<endl;
p = array;
cout<<p<<endl;
cout<<array<<endl;
```

0x7ff7b3ebed00 ← p的内存的首地址

0x7ff7b3ebed10 ← p的内存中所保存的数组首地址

0x7ff7b3ebed10 ← 数组首地址

```
int main(){
int array[9]={1, 2, 3, 4, 5, 6, 7, 8, 9};
int *p, *q;
int a = 10;
char ch[3] = {'a','b'};
q = &a;

cout<<ch<<endl;
cout<<"array: "<<array<<endl;
cout<<"&array: "<<&array<<endl;
cout<<"array+1: "<<array+1<<endl; //+4 bytes
cout<<&array + 2<<endl; //72 bytes more

cout<<&p<<endl;
p = array;
cout<<"*p: "<<*p<<endl;
cout<<"array: "<<array<<endl;
cout<<"*(array+1): "<<*(array+1)<<endl;

cout<<q<<endl;
cout<<*(q+1)<<endl; //出来的是混乱的值，因为访问到了非法内存
cout<<*(p+1)<<endl; //对于指向数组头的指针，按照数组头方式操作
return 0;
}
```



代码演示

chapter7-pointer-cast.cpp

```
ab      字符串指针
array: 0x7ff7b3e56d20   数组头地址
&array: 0x7ff7b3e56d20   整个数组的地址
array+1: 0x7ff7b3e56d24 数组下一个元素
0x7ff7b3e56d68   数组整体地址+2: 2*9*4 bytes
0x7ff7b3e56d10   p地址
*p: 1    p所指元素的值
array: 0x7ff7b3e56d20 数组头
*(array+1): 2   数组中第一个元素的值
0x7ff7b3e56d04   变量a的地址
-1276809980   变量a的地址+1中所存的元素的值
2    p+1中所存的元素的值
```

数组元素的指针表示

数组名是指针

- array[k]地址是 array + k
- array[k]等价于 *(array + k)

执行 intptr= array 后

- array[k]地址是 intptr + k
- array[k]等价于 *(intptr + k)
- array[k]等价于 intptr[k]

小练习

以下哪些表达式能获取int数组a中元素a[1]的地址？
【多选题】

A: &a[1]

B: a+1

C: &a+1

D: a+4

小练习

以下哪些表达式能获取int数组a中元素a[1]的地址？
【多选题】

A：&a[1]

B：a+1

C：&a+1

D：a+4

答案： A 和 B（数组名本身已是地址，&a不符合语法）

本讲大纲

- **指针概念及指针变量的定义**
- **指针运算**
- **指针与数组**
- **动态内存分配**
- **字符串与指针**

动态内存分配

- 在C++语言中，每个程序需要用到几个变量，在写程序前就应该知道。每个数组有几个元素也必须在写程序时就决定
- 有时在程序开始运行前我们**并不知道**我们需要多大的数组元素，因此希望能在程序中根据某一个当前运行值来决定数组的大小。
  - 如设计一个打印魔阵的程序，我们希望先输入魔阵的阶数，然后根据阶数定义一个矩阵。

动态内存分配

- 程序运行过程中申请变量，可以是简单变量或数组

- **实现方法**
  - 定义一个指针
  - 申请一块内存，地址存入指针
  - 通过指针间接访问动态申请的内存
  - 如：

```
int *scores;
scores = 内存的起始地址；
```

动态内存分配与回收

- 用C语言编程，动态分配是通过系统函数malloc(),free()和运算符sizeof来实现动态分配。
- C++中由 new 和 delete 两个运算符替代。
- 运算符new用于进行内存分配：
  - p = new type;
  - p = new type[size];//分配数组
  - p = new type(初值)；
- 运算符delete释放new分配的内存：
  - delete p;
  - delete [] p;//数组的情况

动态内存分配与回收

申请动态变量

申请动态变量：p = new type;

申请动态数组：p = new type[size];

申请动态变量并初始化：p = new type(初值);

申请动态数组并初始化：p = new int[5]{1,2,3,4,5};

释放动态变量的空间

动态变量的空间必须由程序释放

释放动态变量：delete p;

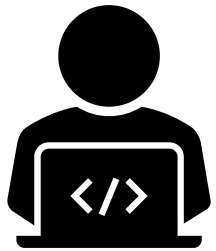
释放动态数组：delete [] p;

字符数组可以不加方括号

没有释放动态变量的空间

内存泄漏

动态内存分配与回收



代码演示

chapter7-dynamic-memory.cpp

```
99abcde
0x7fa24e705bd0  0x7fa24e705be0
0x7ff7b3c1cd50  0x7ff7b3c1cd48
```

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

int main(){
        int *p;
        char *q;

        p = new int(99); //动态分配内存，并将99作为初始化值赋给它
        q = new char[10];
        strcpy(q, "abcde");
        cout<< *p << q << endl;
        cout << p << '\t' << (void *) q << endl;
        cout << &p << '\t' << &q << endl;
        delete p;
        delete q;

        return 0;
}
```

拓展：试试看分析这段代码的输出

现象1: 字符类型数组q直接输出字符串本身。
解析：
- cout 对 char* 有**特殊处理** —— 会从首地址开始输出字符，直到遇到 '\0'（字符串结束符）

现象2: p和&p是不一样的。
解析：
- p 是指针变量，它**存储的内容**是动态内存的首地址
- &p 是指针变量 p 自己的内存地址。

现象3: (void *)q和&q是不一样的。
解析：
- (void *)q将char*强制转换为void*，cout 对 void* 会输出**指针存储的地址**（即字符数组的首地址）
- &q输出指针变量 q **自身的内存地址**（与&p同理）

```
p = new int(99);
q = new char[10];
strcpy(q, "abcde");
// int a[3] = {1, 2, 3};
// char b[4] = {'a','b','c'};

cout<< *p << '\t' << q << endl;
// cout << a << '\t' << b << endl;

cout << p << '\t' << (void *) q << endl;
cout << &p << '\t' << &q << endl;
cout << *(&p) << '\t' << *(&q) << endl;
delete p;
delete q;
```

```
99      abcde
0x7fded1f05f70  0x7fded1f060c0
0x7ff7b491cd30  0x7ff7b491cd28
0x7fded1f05f70  abcde
```

拓展：进一步解析下面的例子

```
//你会看到下面代码中，(void *)q和&q是一样的：
char q[3] = {'a', 'b'};  // 字符数组，存储'a','b'，第三个元素默认'\0'（因为初始化列表长度不足）
cout << q << endl;       // 输出从q[0]开始的字符串（直到遇到'\0'），即"ab"
cout << (void *)q << endl;  // 强制转换为void*，输出数组q的首地址（q[0]的地址）
cout << &q << endl;      // 输出数组q整体的地址（数组作为一个整体的地址）
```

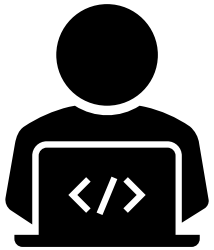
```
ab
0x16b36e76c
0x16b36e76c
```

现象：(void *)q和&q是一样的。
解析：

- 输出q则输出从q[0]开始的字符串（直到遇到'\0'），即"ab"
- 输出(void *)q则强制转换为void*，输出数组q的首地址（q[0]的地址）
- 输出&q则输出数组q整体的地址（数组作为一个整体的地址）

检查动态分配是否成功：手动检查

- new 操作的结果是申请到的空间的地址
- 当系统空间用完时，new 操作可能失败
- new 操作失败时，返回空指针
- new操作后最好检查一下是否成功



代码演示

chapter7-dynamic-memory.cpp

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

int main()
{
	int *p;
	p = new int;
	if ( !p ) {
		cout << "allocation failure\n";
		return 1;
	}
	*p = 20;
	cout << *p;
	delete p;

	return 0;
}
```

assert宏

assert（）宏在标准头文件cassert中

格式

assert(逻辑表达式)

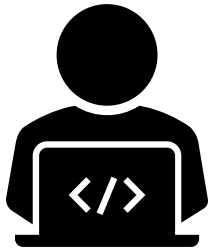
作用

如果表达式为假，则在发出一个错误消息后程序会终止

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

int main()
{
    int *p;

    p = new int;
    assert (p != 0)；//若p等于NULL，则退出程序执行
    *p=20;
    cout << *p;
    delete p;

    return 0;
}
```

检查动态分配是否成功：Try and catch

- 后续面向对象程序设计中涉及

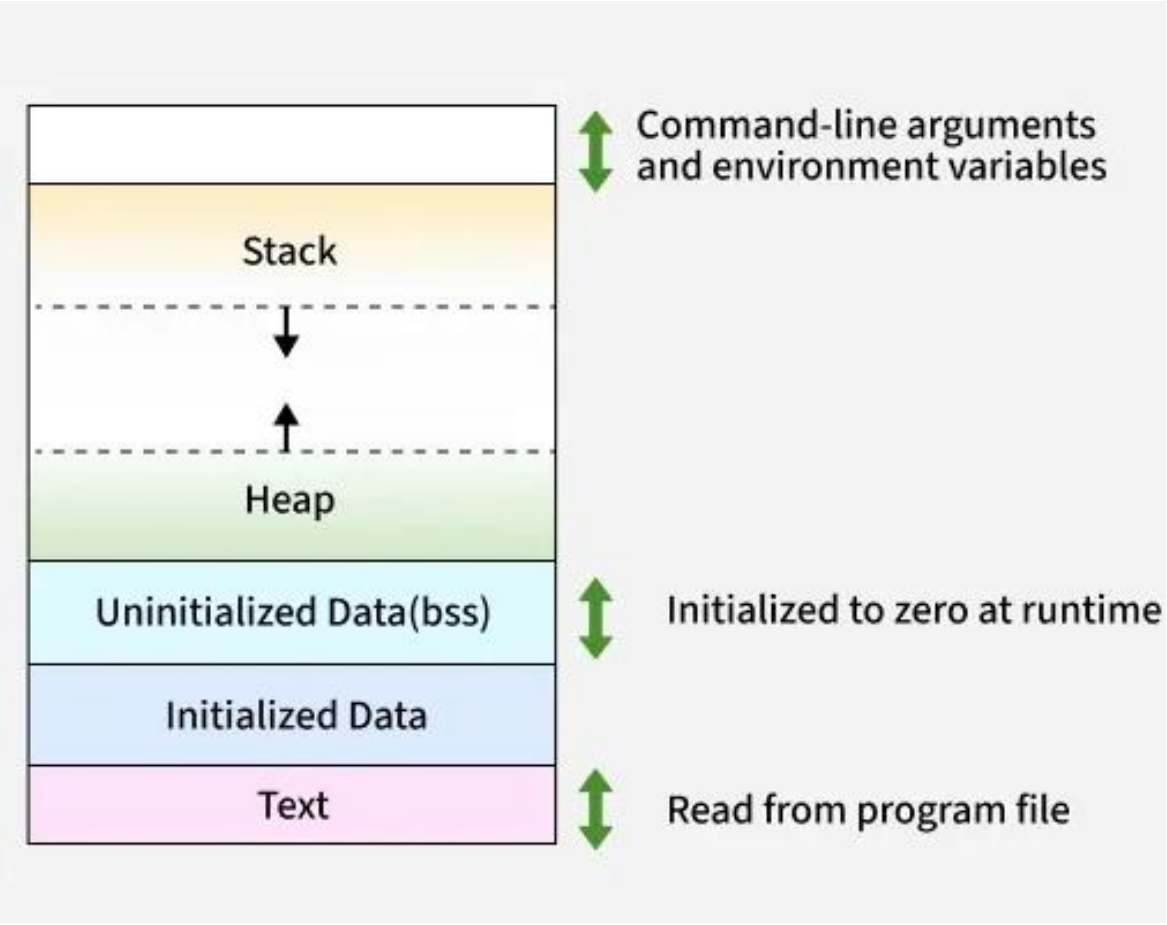


代码演示

chapter7-memory-allocation-failure.cpp

```cpp
// Driver Code
int main()
{
        // Allocate huge amount of memory
        long MEMORY_SIZE = 0x7fffffff;

        // Put memory allocation statement
        // in the try catch block
        try {
                char* ptr = new char[MEMORY_SIZE];

                // When memory allocation fails,
                // below line is not be executed
                // & control will go in catch block
                cout << "Memory is allocated"
                << " Successfully" << endl;
                }

        // Catch Block handle error
        catch (const bad_alloc& e) {

                cout << "Memory Allocation"
                << " is failed: "
                << e.what()
                << endl;
        }

return 0;
}
```

内存分配的进一步介绍



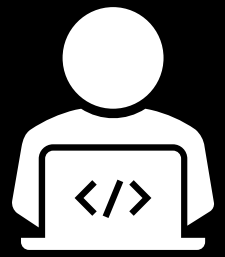
堆动态分配：在程序执行过程中需要新的存储空间时，可用动态分配的方法向系统申请新的空间，当不再使用时用显式的方法还给系统（非自动）。这部分空间是从被称为堆(heap)的内存区域分配的。

栈自动变量. 函数被调用时，空间被分配；函数执行结束后，空间被释放

动态变量应用示例

- 输入一批数据，计算它们的和。数据个数在设计程序时还无法确定
- 存储一批数据（同类）应该用数组，但C++的数组大小必须是固定的。该问题有两个解决方案：
  - 开设一个足够大的数组，每次运行时只使用一部分。缺点：浪费空间
  - 用动态内存分配根据输入的数据量申请一个动态数组

```cpp
#include <iostream>
using namespace std;
int main() {
	int *p, i, n, sum=0;
	cout << "An array will be created dynamically.\n" ;
	cout << "Input an array size n followed by n integers:";
	cin >> n;

	p = new int [n];
	assert( p != NULL);

	for (i=0; i<n; ++i) cin >> p[i];
	for(i=0; i<n; ++i) sum += p[i];
	delete [] p;
	cout << "Number of elements:" << n << endl;
	cout << "Sum of the elements:" << sum << endl;
	return 0;
}
```



代码演示

chapter7-dynamic- memory-example.cpp

本讲大纲

- **指针概念及指针变量的定义**
- **指针运算**
- **指针与数组**
- **动态内存分配**
- **字符串与指针**

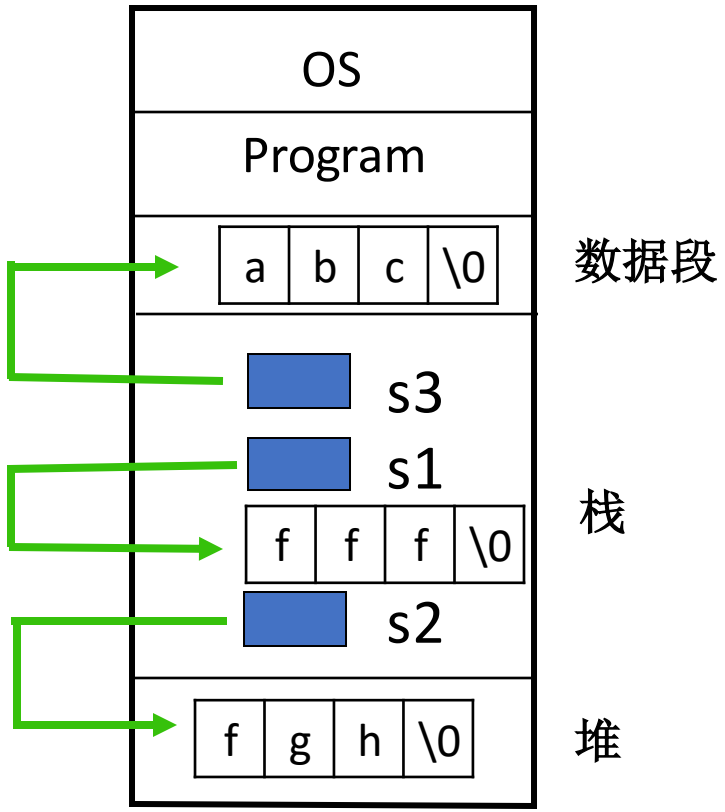
字符串与指针

- 字符串的另一种表示是定义一个指向字符的指针，然后直接将一个字符串常量赋给它
- 如：
  - char *string;
  - string = “abcde”;
- 将存储字符串“abcde”的内存的首地址赋给指针变量string
- 字符串常量存储在一个称为数据段的内存区域里

用指针表示字符串的其他用法

- 将表示字符串的字符数组名赋给指针。
- 用动态内存分配的方法申请一块空间存储字符串，让指针指向这块空间。

```
int main() {
    char  *s1, *s2, *s3 = “abc”;
    char  ch[] = “fff”;
    s1 = ch; s2 = new char[4];
    strcpy(s2, “fgh”);
    return 0;
}
```



字符串作为函数的参数

- 字符串作为参数和数组名作为参数的传递一样，可以有两种方法：
  - 作为字符数组传递
  - 作为指向字符的指针传递
- 两种传递方式的本质是一样的，都是传递了字符串的首地址。
- 和一般数组不同，字符串作为字符数组传递时不需要指定长度。因为字符串操作的结束是有依据的，即‘¥0’。

const与指针

const int *p;

不能通过p修改它指向的单元的内容，但p可以指向不同的变量

int *const p = &x;

可以通过p修改指向的对象，但p的值不能变

即p永远只能指向x。p定义时必须给出初值

const int *const p = &x;

p的值不能修改，*p的值也不能修改

提高安全性

超长加法程序

- 试设计一个函数，计算两个128位的整数（十进制）的和，结果作为函数值返回。
  - 设计思想：用一个由数字组成的字符串来表示数据。如允许的最大长度为128位，则字符数组的长度为128。
  - 如数字123，可表示为：

127	…	5	4	3	2	1	0
‘0’	‘0’	‘0’	‘0’	‘0’	‘1’	‘2’	‘3’

思路：set(“123”, num1); //即把用户定义的任意长短的数据用字符串来表示

超长加法程序：生成函数设计

```
int set(const char *s1, char *s) { //从字符串常量到数组/指针
    int i= strlen(s1) - 1, j = 0;

    while (i>= 0) {
        if (s1[i] > '9' || s1[i] < '0')
        { cout << "set error\n";  return -1; }
        else { s[j] = s1[i]; --i; ++j; }
    }
    for ( ; j < 128; ++j ) s[j] =  ‘0’ ;   // 前置空格

    return 0;
}
```

超长加法程序：加法函数设计

```
void add(const char  *p1, const char  *p2, char *s)  {
    int i, j = 0; /* j为进位 */

    for (i=0; i < 128; ++i)  {
        s[i] = p1[i] - '0' + p2[i] - '0' + j;
        if ( s[i] >= 10)  j = s[i] / 10; else j = 0;
        s[i] = s[i] % 10 + '0';
    }
}
```

超长加法程序：显示函数设计

```
void show(const char *s) {
    int i=127;

    while (s[i] == '0')  --i; // 跳过前置的0
    while (i>=0) {
        cout << s[i];
        --i;
    }
    cout << endl;
}
```

超长加法程序：示例

```
int main() {
    char num1[128], num2[128], sum[128];

    set("123", num1);
    set("345", num2);

    show(num1);
    show(num2);
    add(num1, num2, sum);
    show(sum);

    return 0;
}
```

执行结果：

123

345

468