

# 第二课：指针和引用

主讲人：陈笑沙

## 本节内容

- 2.1 指针概念
- 2.2 指针运算
- 2.3 指针与数组
- 2.4 堆内存分配
- 2.5 const 指针
- 2.6 指针与函数
- 2.7 字符串指针
- 2.8 命令行参数
- 2.9 引用的概念
- 2.10 左值与右值
- 2.11 const 引用

## 2.1 指针的概念

如何定义？

```
int a = 3;  
int *p;  
int *pa = &a;
```

# 2.1 指针的概念

## 指针的意义

# 2.1 指针的概念

## 指针的意义

- 从数值的访问走向地址的访问(地址值的数值化)

# 2.1 指针的概念

## 指针的意义

- 从数值的访问走向地址的访问(地址值的数值化)
- 从访问名字所在存储位置走向访问任何存储位置

## 2.1 指针的概念

### 指针的意义

- 从数值的访问走向地址的访问(地址值的数值化)
- 从访问名字所在存储位置走向访问任何存储位置
- 获得高效数据访问的同时,也带来数据的不安全性

# 2.1 指针的概念

## 指针的类型



# 2.1 指针的概念

## 指针的类型

- 定义时星号前的类型即指针类型

# 2.1 指针的概念

## 指针的类型

- 定义时星号前的类型即指针类型
- 一定类型的指针指向一定类型的实体

# 2.1 指针的概念

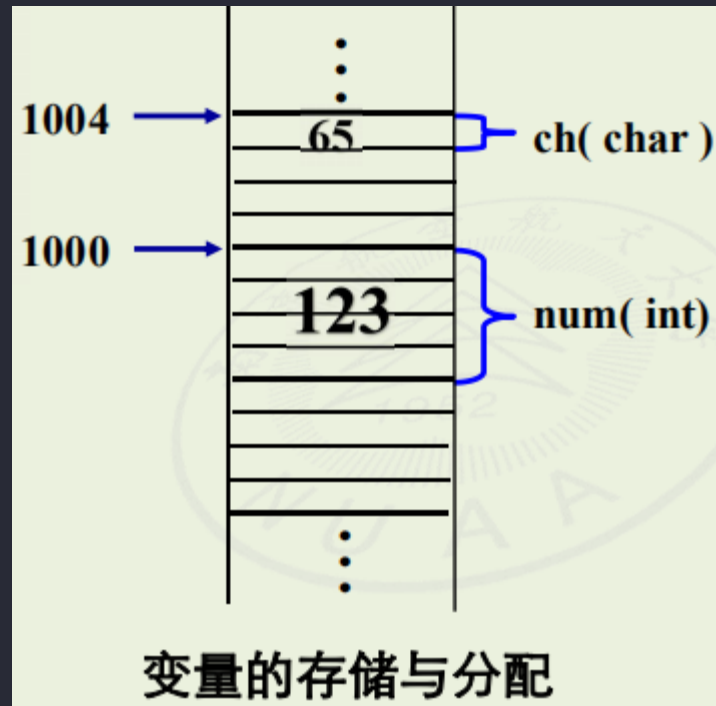
## 指针的类型

- 定义时星号前的类型即指针类型
- 一定类型的指针指向一定类型的实体
- 指针类型是：
  - 指针操作的依据
  - 编译检查的依据

## 2.1 指针的概念

假设定义如下变量

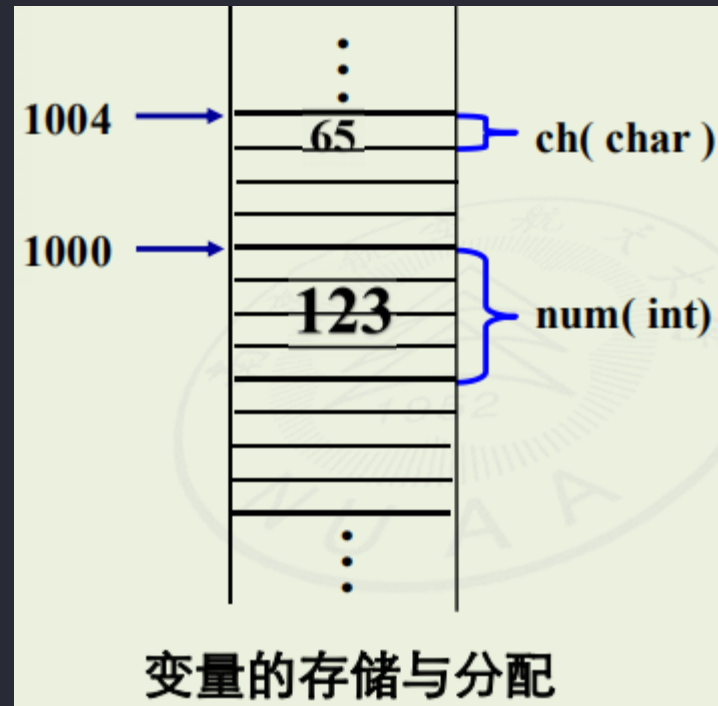
```
int num = 123;  
char ch = 'A';
```



## 2.1 指针的概念

取内存地址操作为

```
int num = 123;  
char ch = 'A';  
  
char *pchar;  
pchar = &ch;
```



# 2.1 指针的概念

example/lec02/pointerBasic

```
#include <iostream>
int main() {
    int a = 100, b = 10;
    int *pointer_1, *pointer_2;

    pointer_1 = &a;
    pointer_2 = &b;
    std::cout << "a = " << a << ", b = " << b << std::endl;
    std::cout << "*pointer_1 = " << *pointer_1
              << ", *pointer_2 = " << *pointer_2 << std::endl;

    return 0;
}
```

# 2.1 指针的概念

如何定义指针变量？

# 2.1 指针的概念

如何定义指针变量？

- 需要基类型



# 2.1 指针的概念

如何定义指针变量？

- 需要基类型
- 在定义指针变量时要注意:

# 2.1 指针的概念

## 如何定义指针变量？

- 需要基类型
- 在定义指针变量时要注意:
  - 指针变量前面的“\*”表示该变量为指针型变量。指针变量名则不包含“\*”。

# 2.1 指针的概念

## 如何定义指针变量？

- 需要基类型
- 在定义指针变量时要注意：
  - 指针变量前面的“\*”表示该变量为指针型变量。指针变量名则不包含“\*”。
  - 在定义指针变量时必须指定基类型。

# 2.1 指针的概念

## 如何定义指针变量？

- 需要基类型
- 在定义指针变量时要注意：
  - 指针变量前面的“\*”表示该变量为指针型变量。指针变量名则不包含“\*”。
  - 在定义指针变量时必须指定基类型。
  - 指针变量中只能存放地址（指针），不要将一个整数赋给一个指针变量。

## 2.1 指针的概念

思考 \* 符号的含义：

```
1 int num = 16, *pNum = &num;  
2 *pNum = 123;  
3 num *= 123;
```

## 2.1 指针的概念

思考 \* 符号的含义：

```
1 int num = 16, *pNum = &num;  
2 *pNum = 123;  
3 num *= 123;
```

## 2.1 指针的概念

思考 \* 符号的含义:

```
1 int num = 16, *pNum = &num;  
2 *pNum = 123;  
3 num *= 123;
```

# 2.2 指针运算

## 算术运算



## 2.2 指针运算

### 算术运算

- 指针加减整数

# 2.2 指针运算

## 算术运算

- 指针加减整数
- 指针递增/递减

## 2.2 指针运算

### 算术运算

- 指针加减整数
- 指针递增/递减
- 指针的步长（与数据类型的关系）

# 2.2 指针运算

## 比较运算

# 2.2 指针运算

## 比较运算

- 指针的大小比较
  - >
  - <
  - <=
  - >=

# 2.2 指针运算

## 比较运算

- 指针的大小比较
  - >
  - <
  - <=
  - >=
- 指针相等性判断
  - ==
  - !=

## 2.2 指针运算

### 指针的差值

- 计算两个指针之间的距离

```
int diff = ptr2 - ptr2;
```

## 2.3 指针与数组

数组**大多数**情况都会退化为指针：

```
int a[3] = {1, 2, 3};  
int *arr = a;  
int *p = &a[0];
```



## 2.3 指针与数组

- 可以有：  $p + 1$ 、 $p += 1$ 、 $p++$ 、 $++p$  等操作。
- 当然也有对应的减法操作。

example/lec02/ptrCalc

```
int a[3] = {1, 2, 3};
int *arr = a;
int *p = &a[0];

cout << p << " " << arr << " " << *arr << endl;
arr += 1;
cout << *arr << endl;

cout << *p << " " << *p++ << endl;
cout << *++p << endl;
```

## 2.3 指针与数组

可以通过指针引用数组元素：

```
int a[] = {1, 2, 3, 4};  
int *p = a;  
  
cout << *(p + 2) << endl;  
cout << p[2] << endl;
```

## 2.3 指针与数组

函数接收数组参数：

```
void printArray(const int* arr, size_t n)
{
    for (int i = 0; i < n; i++)
    {
        cout << arr[i];
        if (i < n - 1)
            cout << ", ";
    }
    cout << endl;
}
```

# 2.3 指针与数组

## 扩展内容

多维数组: <example/lec02/multiDim>

```
int a[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
// type of a: int[3][4], sometimes int(*)[4]
// type of a[0]: int[4], sometimes int*
// type of a[0][0]: int
cout << "sizeof(a): " << sizeof(a) / sizeof(int) << endl;

// int **p = a; wrong
int (*p)[4] = a;
cout << "sizeof(*p): " << sizeof(*p) / sizeof(int) << endl;

cout << "*(p + 1)[0]: " << *(p + 1)[0] << endl;
cout << "**(p + 1): " << **(p + 1) << endl;
cout << "p[2][1]: " << p[2][1] << endl;
cout << "*(*(p + 2) + 1): " << (*(p + 2) + 1) << endl;
```

## 2.3 指针与数组

### 扩展内容

特例：对于编译期间确定大小的数组，可以用模板捕获其大小。

```
template<size_t N>
void printArray(const int (&arr)[N])
{
    for (int i = 0; i < N; i++)
    {
        cout << arr[i];
        if (i < N - 1)
            cout << ", ";
    }
    cout << endl;
}
```

## 2.3 指针与数组

C++ 版替代方案: `vector`

`example/lec02/primes`

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int main(int argc, char** argv) {
7     if (argc < 2) {
8         cout << "Usage: " << argv[0] << " [max number]" << endl;
9         return EXIT_FAILURE;
10    }
11    int maxNum = atoi(argv[1]);
12    if (maxNum < 2) {
13        return 0;
14    }
15
16    vector<bool> primes(maxNum + 1, true);
```

# 2.3 指针与数组

C++ 版替代方案: `vector`

`example/lec02/primes`

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int main(int argc, char** argv) {
7     if (argc < 2) {
8         cout << "Usage: " << argv[0] << " [max number]" << endl;
9         return EXIT_FAILURE;
10    }
11    int maxNum = atoi(argv[1]);
12    if (maxNum < 2) {
13        return 0;
14    }
15
16    vector<bool> primes(maxNum + 1, true);
```

# 2.3 指针与数组

C++ 版替代方案: `vector`

`example/lec02/primes`

```
1  #include <vector>
2
3
4  using namespace std;
5
6  int main(int argc, char** argv) {
7      if (argc < 2) {
8          cout << "Usage: " << argv[0] << " [max number]" << endl;
9          return EXIT_FAILURE;
10     }
11     int maxNum = atoi(argv[1]);
12     if (maxNum < 2) {
13         return 0;
14     }
15
16     vector<bool> primes(maxNum + 1, true);
16     vector<bool> primes(maxNum + 1, true);
```



# 2.3 指针与数组

C++ 版替代方案: `vector`

`example/lec02/primes`

```
5
6 int main(int argc, char** argv) {
7     if (argc < 2) {
8         cout << "Usage: " << argv[0] << " [max number]" << endl;
9         return EXIT_FAILURE;
10    }
11    int maxNum = atoi(argv[1]);
12    if (maxNum < 2) {
13        return 0;
14    }
15
16    vector<bool> primes(maxNum + 1, true);
17    for (int i = 2; i < primes.size() - 1; i++)
18    {
19        for (int j = i + 1; j < primes.size(); j++) {
20            if (!primes[j])
21                continue;
22            if (j % i == 0) {
23                primes[j] = false;
24            }
25        }
26    }
27    for (int i = 2; i < primes.size(); i++) {
28        if (primes[i]) {
29            cout << i << " ";
30            if (i % 10 == 0) {
31                cout << endl;
32            }
33        }
34    }
35    cout << endl;
36    return 0;
37 }
```

# 2.3 指针与数组

C++ 版替代方案: `vector`

`example/lec02/primes`

```
9         return EXIT_FAILURE;
10     }
11     int maxNum = atoi(argv[1]);
12     if (maxNum < 2) {
13         return 0;
14     }
15
16     vector<bool> primes(maxNum + 1, true);
17     for (int i = 2; i < primes.size() - 1; i++)
18     {
19         for (int j = i + 1; j < primes.size(); j++) {
20             if (!primes[j])
21                 continue;
22             if (j % i == 0)
23                 primes[j] = false;
24         }
25     }
26     vector<bool> primes(maxNum + 1, true);
```

# 2.3 指针与数组

C++ 版替代方案: `vector`

`example/lec02/primes`

```
14     }
15
16     vector<bool> primes(maxNum + 1, true);
17     for (int i = 2; i < primes.size() - 1; i++)
18     {
19         for (int j = i + 1; j < primes.size(); j++) {
20             if (!primes[j])
21                 continue;
22             if (j % i == 0)
23                 primes[j] = false;
24         }
25     }
26
27     for (int i = 2; i < maxNum; i++)
28         if (primes[i])
16     vector<bool> primes(maxNum + 1, true);
```

## 2.3 指针与数组

C++ 版替代方案: `vector`

`example/lec02/primes`

```
18     {
19         for (int j = i + 1; j < primes.size(); j++) {
20             if (!primes[j])
21                 continue;
22             if (j % i == 0)
23                 primes[j] = false;
24         }
25     }
26
27     for (int i = 2; i < maxNum; i++)
28         if (primes[i])
29             cout << i << " ";
30     cout << endl;
31
32     return 0;
33 }
16 vector<bool> primes(maxNum + 1, true);
```

# 2.4 堆内存分配

## C 语言方法

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int size = 4, i;
7     int *p = (int*)malloc(size * sizeof(int));
8     for (i = 0; i < size; i++)
9         p[i] = i + 1;
10    for (i = 0; i < size; i++)
11        printf("%d ", p[i]);
12
13    free(p);
14    return 0;
15 }
```

# 2.4 堆内存分配

## C 语言方法

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int size = 4, i;
7     int *p = (int*)malloc(size * sizeof(int));
8     for (i = 0; i < size; i++)
9         p[i] = i + 1;
10    for (i = 0; i < size; i++)
11        printf("%d ", p[i]);
12
13    free(p);
14    return 0;
15 }
```

# 2.4 堆内存分配

## C 语言方法

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int size = 4, i;
7     int *p = (int*)malloc(size * sizeof(int));
8     for (i = 0; i < size; i++)
9         p[i] = i + 1;
10    for (i = 0; i < size; i++)
11        printf("%d ", p[i]);
12
13    free(p);
14    return 0;
15 }
```

# 2.4 堆内存分配

## C 语言方法

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int size = 4, i;
7     int *p = (int*)malloc(size * sizeof(int));
8     for (i = 0; i < size; i++)
9         p[i] = i + 1;
10    for (i = 0; i < size; i++)
11        printf("%d ", p[i]);
12
13    free(p);
14    return 0;
15 }
```



# 2.4 堆内存分配

## C++ 语言方法

```
1 #include <iostream>
2
3 int main()
4 {
5     int size = 4;
6     int *p = new int[size];
7     for (int i = 0; i < size; i++)
8         p[i] = i + 1;
9     for (int i = 0; i < size; i++)
10         std::cout << p[i] << " ";
11
12     delete[] p;
13     return 0;
14 }
```

# 2.4 堆内存分配

## C++ 语言方法

```
1 #include <iostream>
2
3 int main()
4 {
5     int size = 4;
6     int *p = new int[size];
7     for (int i = 0; i < size; i++)
8         p[i] = i + 1;
9     for (int i = 0; i < size; i++)
10         std::cout << p[i] << " ";
11
12     delete[] p;
13     return 0;
14 }
```

# 2.4 堆内存分配

## C++ 语言方法

```
1 #include <iostream>
2
3 int main()
4 {
5     int size = 4;
6     int *p = new int[size];
7     for (int i = 0; i < size; i++)
8         p[i] = i + 1;
9     for (int i = 0; i < size; i++)
10         std::cout << p[i] << " ";
11
12     delete[] p;
13     return 0;
14 }
```

# 2.4 堆内存分配

## C++ 语言方法

```
1 #include <iostream>
2
3 int main()
4 {
5     int size = 4;
6     int *p = new int[size];
7     for (int i = 0; i < size; i++)
8         p[i] = i + 1;
9     for (int i = 0; i < size; i++)
10         std::cout << p[i] << " ";
11
12     delete[] p;
13     return 0;
14 }
```

## 2.4 堆内存分配

`new` 与 `delete` 还有其他用法，涉及到类，之后再讲。

# 2.4 堆内存分配

C++ 替代方案：智能指针：

example/lec02/pascalTrig

```
1 #include <iostream>
2 #include <memory>
3 #include <vector>
4
5 using namespace std;
6
7 int main(int argc, char **argv)
8 {
9     // in this example
10    // pure vector is better
11    int n = 10;
12    vector<unique_ptr<int[]>> pascalTrig(n);
13
14    for (int i = 0; i < n; i++)
15    {
16        pascalTrig[i] = make_unique<int[]>(i + 1);
```

# 2.4 堆内存分配

C++ 替代方案：智能指针：

example/lec02/pascalTrig

```
1 #include <iostream>
2 #include <memory>
3 #include <vector>
4
5 using namespace std;
6
7 int main(int argc, char **argv)
8 {
9     // in this example
10    // pure vector is better
11    int n = 10;
12    vector<unique_ptr<int[]>> pascalTrig(n);
13
14    for (int i = 0; i < n; i++)
15    {
16        pascalTrig[i] = make_unique<int[]>(i + 1);
```

# 2.4 堆内存分配

C++ 替代方案：智能指针：

example/lec02/pascalTrig

```
5 using namespace std;
6
7 int main(int argc, char **argv)
8 {
9     // in this example
10    // pure vector is better
11    int n = 10;
12    vector<unique_ptr<int[]>> pascalTrig(n);
13
14    for (int i = 0; i < n; i++)
15    {
16        pascalTrig[i] = make_unique<int[]>(i + 1);
17        pascalTrig[i][0] = 1;
18
19        if (i == 0)
20            pascalTrig[i+1] = make_unique<int[]>(i + 1);
```



# 2.4 堆内存分配

C++ 替代方案：智能指针：

example/lec02/pascalTrig

```
9 // in this example
10 // pure vector is better
11 int n = 10;
12 vector<unique_ptr<int[]>> pascalTrig(n);
13
14 for (int i = 0; i < n; i++)
15 {
16     pascalTrig[i] = make_unique<int[]>(i + 1);
17     pascalTrig[i][0] = 1;
18
19     if (i == 0)
20         continue;
21     for (int j = 1; j <= i; j++)
22     {
23         if (j == i)
24             pascalTrig[i][j] = make_unique<int[]>(1 + 1);
```

# 2.4 堆内存分配

C++ 替代方案：智能指针：

example/lec02/pascalTrig

```
20         continue;
21     for (int j = 1; j <= i; j++)
22     {
23         if (j == i)
24         {
25             pascalTrig[i][j] = 1;
26         }
27         else
28         {
29             pascalTrig[i][j] =
30                 pascalTrig[i - 1][j - 1] +
31                 pascalTrig[i - 1][j];
32         }
33     }
34 }
16 pascalTrig111 = make_unique<int**>(1 + 1);
```

# 2.4 堆内存分配

C++ 替代方案：智能指针：

example/lec02/pascalTrig

```
32     }  
33     }  
34 }  
35  
36 for (int i = 0; i < n; i++)  
37 {  
38     for (int j = 0; j <= i; j++)  
39     {  
40         cout << pascalTrig[i][j] << "\t";  
41     }  
42     cout << "\n";  
43 }  
44 cout << endl;  
45  
46 return 0;  
47 }  
16     pascalTrig111 = make_unique<int**>(1 + 1);
```

## 2.5 `const` 指针

`const` 的基本作用

## 2.5 `const` 指针

`const` 的基本作用

- 保护数据不被意外修改

## 2.5 `const` 指针

### `const` 的基本作用

- 保护数据不被意外修改
- 提高代码可读性

## 2.5 `const` 指针

### `const` 的基本作用

- 保护数据不被意外修改
- 提高代码可读性
- 编译器优化机会

## 2.5 const 指针

### const 的基本作用

- 保护数据不被意外修改
- 提高代码可读性
- 编译器优化机会

```
const int size = 100; // 常量声明
```



## 2.5 const 指针

### 指向常量的指针 (Pointer to Constant)

```
const int* ptr; // 指针可变，指向的数据不可变
int const* ptr1; // 和ptr一样

*ptr = 10; // wrong!
```

- 可以修改指针指向的地址
- 不能通过指针修改数据

## 2.5 const 指针

### 常量指针 (Constant Pointer)

```
int* const ptr = &var; // 指针不可变，指向的数据可变
```

- 指针地址固定
- 可以通过指针修改数据

## 2.5 const 指针

指向常量的常量指针

```
const int* const ptr = &var; // 指针和数据都不可变
```

## 2.5 const 指针

形式	指针可 变性	数据可 变性	声明示例
普通指针	✓	✓	<code>int* ptr</code>
指向常量的 指针	✓	✗	<code>const int* ptr</code>
常量指针	✗	✓	<code>int* const ptr</code>
双向const 指针	✗	✗	<code>const int* const ptr</code>

## 2.5 const 指针

```
const char* str1 = "Hello"; // 正确：字符串字面量是常量  
char* str2 = "World";      // 错误：C++11起禁止（需强制转换）
```

## 2.6 指针与函数

函数可以接收指针作为参数，也可以返回指针，但是有以下易犯错误：

```
int* sum(const int* a, const int* b) {  
    int c = *a + *b;  
    return &c; // Wrong!  
    // int* c = new int(*a + *b);  
    // return c; // OK, remember to delete it.  
}
```

## 2.6 指针与函数

### 指向函数的指针

```
// 声明函数原型
int add(int, int);

// 声明函数指针
int (*pf)(int, int) = &add;
int (*pf2)(int, int) = add; // also ok

// 调用
pf(1, 2);
pf2(1, 2);
add(1, 2);
```

## 示例：牛顿迭代法

example/lec02/newton

```
1 #include <iostream>
2 #include <iomanip>
3 #include <cmath>
4
5 using Fn = double (*)(double);
6 // typedef double (*Fn)(double); // old way
7
8 double newton(Fn func, double guess = 1.0)
9 {
10     double y = func(guess);
11     if (std::abs(y) < 1e-10) {
12         return guess;
13     }
14
15     double delta = 1e-10;
16     double dv = (func(guess + delta) - func(guess - delta)) / (2 * c
```



注意：函数指针没有加减运算。

## 示例：牛顿迭代法

example/lec02/newton

```
1 #include <iostream>
2 #include <iomanip>
3 #include <cmath>
4
5 using Fn = double (*)(double);
6 // typedef double(*Fn)(double); // old way
7
8 double newton(Fn func, double guess = 1.0)
9 {
10     double y = func(guess);
11     if (std::abs(y) < 1e-10) {
12         return guess;
13     }
14
15     double delta = 1e-10;
16     double dv = (func(guess + delta) - func(guess - delta)) / (2 * c
```

注意：函数指针没有加减运算。

## 示例：牛顿迭代法

example/lec02/newton

```

4
5 using Fn = double (*)(double);
6 // typedef double(*Fn)(double); // old way
7
8 double newton(Fn func, double guess = 1.0)
9 {
10     double y = func(guess);
11     if (std::abs(y) < 1e-10) {
12         return guess;
13     }
14
15     double delta = 1e-10;
16     double dy = (func(guess + delta) - func(guess - delta)) / (2 * delta);
17     return newton(func, guess - y / dy);
18 }
19
20 double dv = (func(guess + delta) - func(guess - delta)) / (2 * delta);

```

注意：函数指针没有加减运算。

## 示例：牛顿迭代法

example/lec02/newton

```

9 {
10     double y = func(guess);
11     if (std::abs(y) < 1e-10) {
12         return guess;
13     }
14
15     double delta = 1e-10;
16     double dy = (func(guess + delta) - func(guess - delta)) / (2 * delta);
17     return newton(func, guess - y / dy);
18 }
19
20 double equation1(double x) {
21     return x * x - 2;
22 }
23
24 double newton1(double func, double guess) {
16     double dy = (func(guess + delta) - func(guess - delta)) / (2 * delta);

```

注意：函数指针没有加减运算。

## 示例：牛顿迭代法

example/lec02/newton

```
18 }
19
20 double equation1(double x) {
21     return x * x - 2;
22 }
23
24 double equation2(double x) {
25     return tan(x) - 1;
26 }
27
28 int main(int argc, char** argv) {
29     std::cout << std::setprecision(10);
30     std::cout << newton(equation1) << std::endl;
31     std::cout << 4 * newton(equation2) << std::endl;
32     return 0;
33 }
16 double dv = (func(guess + delta) - func(guess - delta)) / (2 * delta);
```



注意：函数指针没有加减运算。

## 示例：牛顿迭代法

example/lec02/newton

```
16     double dy = (func(guess + delta) - func(guess - delta)) / (2 * c
17     return newton(func, guess - y / dy);
18 }
19
20 double equation1(double x) {
21     return x * x - 2;
22 }
23
24 double equation2(double x) {
25     return tan(x) - 1;
26 }
27
28 int main(int argc, char** argv) {
29     std::cout << std::setprecision(10);
30     std::cout << newton(equation1) << std::endl;
16     double dy = (func(guess + delta) - func(guess - delta)) / (2 * c
```

注意：函数指针没有加减运算。

## 2.6 指针与函数

## 2.6 指针与函数

- 函数还可以返回函数指针，但是在 C++ 中用法不多。

## 2.6 指针与函数

- 函数还可以返回函数指针，但是在 C++ 中用法不多。
- 现代 C++ 可以结合 lambda 函数使用。

## 2.6 指针与函数

- 函数还可以返回函数指针，但是在 C++ 中用法不多。
- 现代 C++ 可以结合 lambda 函数使用。
- 实际工程建议使用 `std::function`，而不是直接使用函数指针。

## 2.7 字符串指针

### 基本声明方式

```
const char* str1 = "Hello"; // 推荐：显式常量  
char str2[] = "World";      // 字符数组  
char* str3 = str2;          // 指向数组的指针
```



## 2.7 字符串指针

内存布局（假设地址从0x1000开始）：

地址	内容
0x1000	'H'
0x1001	'e'
0x1002	'l'
0x1003	'l'
0x1004	'o'
0x1005	'\0'

## 2.7 字符串指针

核心特征：

- 以空字符\0结尾
- 字符串字面量存储在只读数据段
- 指针存储的是首字符地址

## 2.7 字符串指针

C++ 中不建议直接使用字符串指针，建议用 `std::string` 替代。

# 2.8 命令行参数

## 标准参数形式

```
int main(int argc, char* argv[]) // 最常用形式
int main(int argc, char** argv)  // 等价写法
```

## 其他合法形式

```
int main() // 无参数版本
int main(void) // C风格明确空参数
```

## 2.8 命令行参数

### 参数含义

参数	名称	描述
argc	Argument Count	命令行参数个数 ( $\geq 1$ )
argv	Argument Vector	参数字符串数组指针

## 2.8 命令行参数

测试: `example/lec02/args`

```
#include <iostream>

int main(int argc, char** argv) {
    for (int i = 0; i < argc; i++) {
        std::cout << argv[i] << std::endl;
    }
    return 0;
}
```

```
xmake run args 1 2 3 4 hello
.\build\windows\x64\release\args.exe 1 2 hello test
./a.out 1 2 3 4
```

## 2.9 引用的概念

### 1.1 基本定义

- **别名机制**：为已存在变量创建新名称
- **必须初始化**：声明时必须绑定到有效对象
- **不可重绑定**：初始化后不能更改目标

```
int x = 10;  
int& ref = x; // ref是x的别名  
ref = 20;     // 修改的是x的值
```

# 2.9 引用的概念

## 核心特性



## 2.9 引用的概念

### 核心特性

- **类型安全**：严格匹配被引用类型

## 2.9 引用的概念

### 核心特性

- **类型安全**：严格匹配被引用类型
- **自动解引用**：使用时不需特殊符号

## 2.9 引用的概念

### 核心特性

- **类型安全**：严格匹配被引用类型
- **自动解引用**：使用时不需特殊符号
- **地址共享**：与被引用对象共享内存地址

## 2.9 引用的概念

### 引用解决的指针问题

指针问题	引用解决方案
空指针风险	必须初始化, 不能为null
野指针问题	绑定后不可改变
内存泄漏隐患	不涉及动态内存管理
语法复杂度高	自动解引用, 语法简洁
双重释放风险	天然遵循RAII <sup>1</sup> 原则

1. RAII: Resource Acquisition Is Initialization, 由 c++ 之父 Bjarne Stroustrup 提出的, 中文翻译为资源获取即初始化

## 2.9 引用的概念

### 引用与指针的区别

特性	引用	指针
初始化要求	必须显式初始化	可延迟初始化
可空性	不可为空	可为 <code>nullptr</code>
重绑定	不可	可改变指向
内存管理	不涉及	需手动分配/释放
语法	自动解引用	需使用 <code>*</code> 和 <code>-&gt;</code> 操作符

# 2.10 左值与右值

## 基本定义

类别	特征	示例
左值	有持久身份，可取地址	变量
右值	临时对象，即将销毁	字面量、表达式结果

## 2.10 左值与右值

## 2.10 左值与右值

- 在引用的别名用法中，只有左值可以创建引用。



## 2.10 左值与右值

- 在引用的别名用法中，只有左值可以创建引用。
- 函数参数可以加两个 **&**，表示右值引用。

## 2.10 左值与右值

- 在引用的别名用法中，只有左值可以创建引用。
- 函数参数可以加两个 **&**，表示右值引用。

```
1 int add(int&&, int&&);
2 int constAdd(const int&, const int&);
3 int main()
4 {
5     int a = 10;
6     std::cout << add(1, 2) << std::endl; // ok
7     std::cout << add(a, 2) << std::endl; // wrong
8     std::cout << constAdd(a, 2) << std::endl; // ok
9
10    return 0;
11 }
```

## 2.10 左值与右值

- 在引用的别名用法中，只有左值可以创建引用。
- 函数参数可以加两个 **&**，表示右值引用。

```
1 int add(int&&, int&&);
2 int constAdd(const int&, const int&);
3 int main()
4 {
5     int a = 10;
6     std::cout << add(1, 2) << std::endl; // ok
7     std::cout << add(a, 2) << std::endl; // wrong
8     std::cout << constAdd(a, 2) << std::endl; // ok
9
10    return 0;
11 }
```

## 2.10 左值与右值

- 在引用的别名用法中，只有左值可以创建引用。
- 函数参数可以加两个 **&**，表示右值引用。

```
1 int add(int&&, int&&);
2 int constAdd(const int&, const int&);
3 int main()
4 {
5     int a = 10;
6     std::cout << add(1, 2) << std::endl; // ok
7     std::cout << add(a, 2) << std::endl; // wrong
8     std::cout << constAdd(a, 2) << std::endl; // ok
9
10    return 0;
11 }
```

## 2.11 const引用

### 核心特性

- **只读访问**：无法通过引用修改原对象
- **延长生命周期**：可绑定到临时对象
- **兼容性**：可接受 `const` 和非 `const` 对象

```
const int& cref = 42; // 合法，延长字面量生命周期
```

# 引用的典型应用

# 引用的典型应用

- 函数参数传递（避免拷贝大对象）

# 引用的典型应用

- 函数参数传递（避免拷贝大对象）
- 返回保护性访问



# 引用的典型应用

- 函数参数传递（避免拷贝大对象）
- 返回保护性访问
- 配合临时对象使用

# 引用的典型应用

## 典型错误：

```
int& add(const int& a, const int& b)
{
    int c = a + b;
    return c; // 错误！返回局部变量的引用
}
```

# 引用的典型应用

求解一元二次方程 (bad way) :

```
bool solve(const double a,
           const double b,
           const double c,
           double& x1,
           double& x2)
{
    double delta = b * b - 4 * a * c;
    if (delta < 0 || a == 0)
        return false;
    double s = sqrt(delta);
    x1 = (-b - s) / (2 * a);
    x2 = (-b + s) / (2 * a);
    return true;
}
```

## 作业：

1. 补完一元二次方程求解程序，分别对以下输入求解并输出结果：
  1.  $a = 1, b = 2, c = 1$
  2.  $a = 1, b = 1, c = 1$
  3.  $a = 1, b = 1, c = -2$
  4.  $a = 0, b = 1, c = 1$
2. 自行学习 `std::pair` 与 `std::optional` 的用法，实现一个求解一元二次方程的函数，并对第 1 问中的输入进行求解与输出（Good way）。

提示：xmake 中需要添加：  
`set_languages("c++17")` 以支持 `optional`。或者使用 `g++` 编译，加上 `--std=c++17` 的参数。

1. <https://zh.cppreference.com/w/cpp/utility/pair>
2. <https://zh.cppreference.com/w/cpp/utility/optional>