LECTURE 2: POINTER AND REFERENCE

Lecturer：陈笑沙

TABLE OF CONTENTS

2.1 POINTER CONCEPT

How to define?

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

2.1 POINTER CONCEPT

The significance of pointers

2.1 POINTER CONCEPT

The significance of pointers

- From accessing numerical values to accessing addresses (numerical representation of address values)

2.1 POINTER CONCEPT

The significance of pointers

- From accessing numerical values to accessing addresses (numerical representation of address values)
- From accessing the storage location of a name to accessing any storage location

2.1 POINTER CONCEPT

The significance of pointers

- From accessing numerical values to accessing addresses (numerical representation of address values)
- From accessing the storage location of a name to accessing any storage location
- Achieving efficient data access while also introducing data insecurity

2.1 POINTER CONCEPT

Pointer types

2.1 POINTER CONCEPT

Pointer types

- The type before the asterisk at definition is the pointer type

2.1 POINTER CONCEPT

Pointer types

- The type before the asterisk at definition is the pointer type
- A pointer of a certain type points to an entity of that type

2.1 POINTER CONCEPT

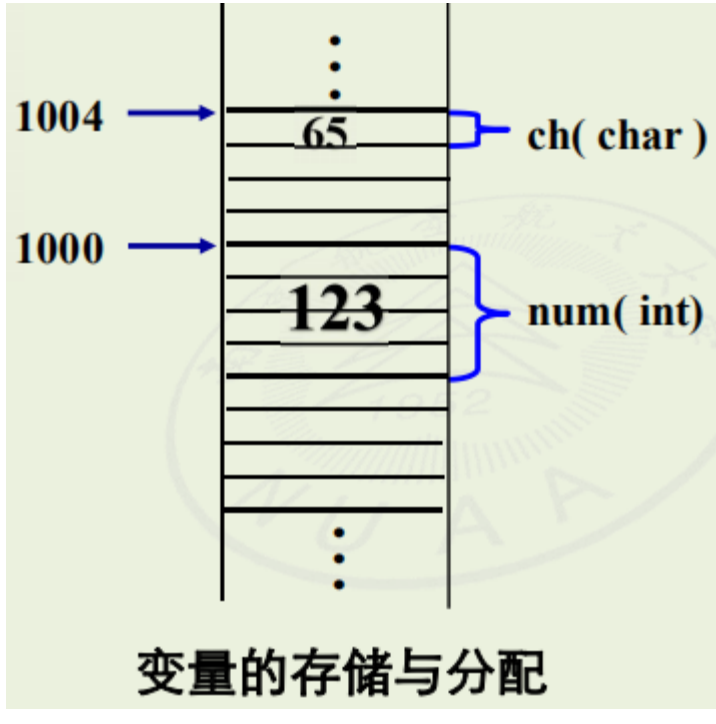
Pointer types

- The type before the asterisk at definition is the pointer type
- A pointer of a certain type points to an entity of that type
- Pointer type is:
  - The basis for pointer operations
  - The basis for compilation checks

2.1 POINTER CONCEPT

If we have following two variables:

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

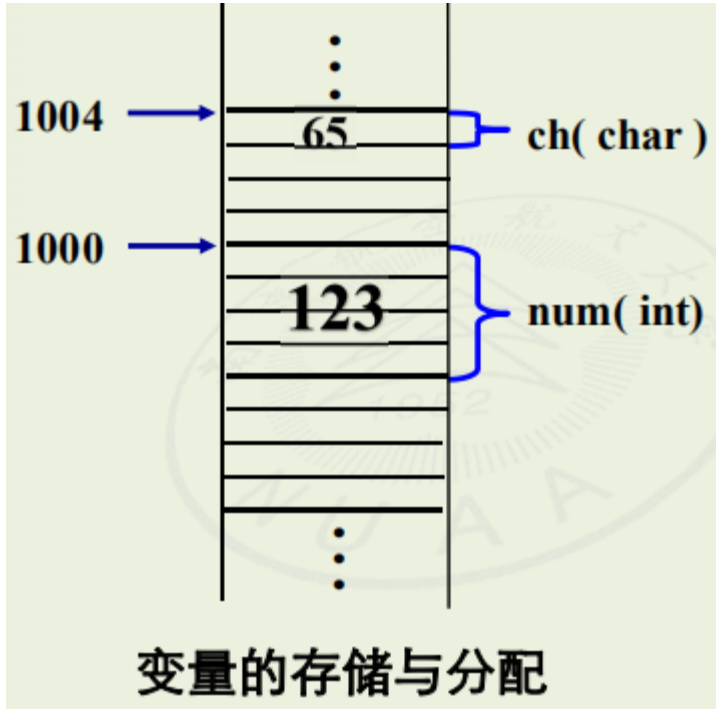


2.1 POINTER CONCEPT

How to get their addresses?

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

char *pchar;
pchar = &ch;
```



2.1 POINTER CONCEPT

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 POINTER CONCEPT

How to define a pointer variable?

2.1 POINTER CONCEPT

How to define a pointer variable?

- Need a base type

2.1 POINTER CONCEPT

How to define a pointer variable?

- Need a base type
- When defining a pointer variable, pay attention to:

2.1 POINTER CONCEPT

How to define a pointer variable?

- Need a base type
- When defining a pointer variable, pay attention to:
  - The "*" before the pointer variable name indicates that the variable is a pointer type variable. The pointer variable name does not contain "*".

2.1 POINTER CONCEPT

How to define a pointer variable?

- Need a base type
- When defining a pointer variable, pay attention to:
  - The "*" before the pointer variable name indicates that the variable is a pointer type variable. The pointer variable name does not contain "*".
  - The base type must be specified when defining a pointer variable.

2.1 POINTER CONCEPT

How to define a pointer variable?

- Need a base type
- When defining a pointer variable, pay attention to:
  - The "*" before the pointer variable name indicates that the variable is a pointer type variable. The pointer variable name does not contain "*".
  - The base type must be specified when defining a pointer variable.
  - A pointer variable can only store addresses (pointers). Do not assign an integer to a pointer variable.

2.1 POINTER CONCEPT

What is the meaning of different asterisks?

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

2.1 POINTER CONCEPT

What is the meaning of different asterisks?

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

2.1 POINTER CONCEPT

What is the meaning of different asterisks?

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

2.2 POINTER OPERATIONS

ARITHMETIC OPERATIONS

2.2 POINTER OPERATIONS

ARITHMETIC OPERATIONS

- Pointer addition and subtraction of integers

2.2 POINTER OPERATIONS

ARITHMETIC OPERATIONS

- Pointer addition and subtraction of integers
- Pointer increment/decrement

2.2 POINTER OPERATIONS

ARITHMETIC OPERATIONS

- Pointer addition and subtraction of integers
- Pointer increment/decrement
- Pointer step size (relationship with data type)

2.2 POINTER OPERATIONS

COMPARISON

2.2 POINTER OPERATIONS

COMPARISON

- Pointer size comparison
  - >
  - <
  - ≤
  - ≥

2.2 POINTER OPERATIONS

COMPARISON

- Pointer size comparison
  - >
  - <
  - ≦
  - ≧
- Pointer equality judgment
  - ==
  - ≠

2.2 POINTER OPERATIONS

POINTER DIFFERENCE

- Calculate the distance between two pointers

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

2.3 POINTERS AND ARRAYS

Array **mostly** will degenerate into pointer:

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

2.3 POINTERS AND ARRAYS

- We have: `p + 1`, `p += 1`, `p++`, `++p` operations.
- Of course, there are corresponding subtraction operations.

`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 POINTERS AND ARRAYS

You can access array elements via pointers:

```
int a[] = {1, 2, 3, 4};
int *p = a;

cout << *(p + 2) << endl;
cout << p[2] << endl;
```

2.3 POINTERS AND ARRAYS

Functions can accept array pointers as arguments:

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

2.3 POINTERS AND ARRAYS

Extended Contents

Multi-Dimension Array： `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 POINTERS AND ARRAYS

Extended Contents

Exception: For arrays whose size is determined at compile time, their size can be captured by the template.

```
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 POINTERS AND ARRAYS

C++ Alternative: `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]" << e
 9         return EXIT_FAILURE;
10     }
11     int maxNum = atoi(argv[1]);
12     if (maxNum < 2) {
13         return 0;
14     }
15
```

2.3 POINTERS AND ARRAYS

C++ Alternative: `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]" << e
 9         return EXIT_FAILURE;
10     }
11     int maxNum = atoi(argv[1]);
12     if (maxNum < 2) {
13         return 0;
14     }
15
```

2.3 POINTERS AND ARRAYS

C++ Alternative: `vector`

`example/lec02/primes`

```
 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]" << e
 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 POINTERS AND ARRAYS

C++ Alternative: `vector`

`example/lec02/primes`

```
 6 int main(int argc, char** argv) {
 7     if (argc < 2) {
 8         cout << "Usage: " << argv[0] << " [max number]" << e
 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         if (!primes[i])
20             continue;
```

2.3 POINTERS AND ARRAYS

C++ Alternative: `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          if (!primes[i])
20              continue;
21          for (int j = i + 1; j < primes.size(); j++) {
22              if (!primes[j])
23                  continue;
```

2.3 POINTERS AND ARRAYS

C++ Alternative: `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          if (!primes[i])
20              continue;
21          for (int j = i + 1; j < primes.size(); j++) {
22              if (!primes[j])
23                  continue;
24              if (j % i == 0)
25                  primes[j] = false;
26          }
27      }
28
```

2.3 POINTERS AND ARRAYS

C++ Alternative: `vector`

`example/lec02/primes`

```
21          for (int j = i + 1; j < primes.size(); j++) {
22              if (!primes[j])
23                  continue;
24              if (j % i == 0)
25                  primes[j] = false;
26          }
27      }
28
29      for (int i = 2; i < maxNum; i++)
30          if (primes[i])
31              cout << i << " ";
32      cout << endl;
33
34      return 0;
35  }
```

2.4 HEAP MEMORY ALLOCATION

C LANGUAGE

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int size = 4, i;
    int *p = (int*)malloc(size * sizeof(int));
    for (i = 0; i < size; i++)
      p[i] = i + 1;
    for (i = 0; i < size; i++)
      printf("%d ", p[i]);

    free(p);
    return 0;
}
```

2.4 HEAP MEMORY ALLOCATION

C LANGUAGE

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int size = 4, i;
    int *p = (int*)malloc(size * sizeof(int));
    for (i = 0; i < size; i++)
      p[i] = i + 1;
    for (i = 0; i < size; i++)
      printf("%d ", p[i]);

    free(p);
    return 0;
}
```

2.4 HEAP MEMORY ALLOCATION

C LANGUAGE

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int size = 4, i;
    int *p = (int*)malloc(size * sizeof(int));
    for (i = 0; i < size; i++)
      p[i] = i + 1;
    for (i = 0; i < size; i++)
      printf("%d ", p[i]);

    free(p);
    return 0;
}
```

2.4 HEAP MEMORY ALLOCATION

C LANGUAGE

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int size = 4, i;
    int *p = (int*)malloc(size * sizeof(int));
    for (i = 0; i < size; i++)
      p[i] = i + 1;
    for (i = 0; i < size; i++)
      printf("%d ", p[i]);

    free(p);
    return 0;
}
```

2.4 HEAP MEMORY ALLOCATION

C LANGUAGE

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int size = 4, i;
    int *p = (int*)malloc(size * sizeof(int));
    for (i = 0; i < size; i++)
      p[i] = i + 1;
    for (i = 0; i < size; i++)
      printf("%d ", p[i]);

    free(p);
    return 0;
}
```

2.4 HEAP MEMORY ALLOCATION

C++ LANGUAGE

```cpp
#include <iostream>

int main()
{
    int size = 4;
    int *p = new int[size];
    for (int i = 0; i < size; i++)
        p[i] = i + 1;
    for (int i = 0; i < size; i++)
        std::cout << p[i] << " ";

    delete[] p;
    return 0;
}
```

2.4 HEAP MEMORY ALLOCATION

C++ LANGUAGE

```cpp
#include <iostream>

int main()
{
    int size = 4;
    int *p = new int[size];
    for (int i = 0; i < size; i++)
        p[i] = i + 1;
    for (int i = 0; i < size; i++)
        std::cout << p[i] << " ";

    delete[] p;
    return 0;
}
```

2.4 HEAP MEMORY ALLOCATION

C++ LANGUAGE

```cpp
#include <iostream>

int main()
{
    int size = 4;
    int *p = new int[size];
    for (int i = 0; i < size; i++)
        p[i] = i + 1;
    for (int i = 0; i < size; i++)
        std::cout << p[i] << " ";

    delete[] p;
    return 0;
}
```

2.4 HEAP MEMORY ALLOCATION

C++ LANGUAGE

```cpp
#include <iostream>

int main()
{
    int size = 4;
    int *p = new int[size];
    for (int i = 0; i < size; i++)
        p[i] = i + 1;
    for (int i = 0; i < size; i++)
        std::cout << p[i] << " ";

    delete[] p;
    return 0;
}
```

2.4 HEAP MEMORY ALLOCATION

C++ LANGUAGE

```cpp
#include <iostream>

int main()
{
    int size = 4;
    int *p = new int[size];
    for (int i = 0; i < size; i++)
        p[i] = i + 1;
    for (int i = 0; i < size; i++)
        std::cout << p[i] << " ";

    delete[] p;
    return 0;
}
```

2.4 HEAP MEMORY ALLOCATION

`new` and `delete` have other uses involving classes, which we'll cover later.

2.4 HEAP MEMORY ALLOCATION

C++ Alternative: Smart Pointers

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     {
```

2.4 HEAP MEMORY ALLOCATION

C++ Alternative: Smart Pointers

example/lec02/pascalTrig

```cpp
#include <iostream>
#include <memory>
#include <vector>

using namespace std;

int main(int argc, char **argv)
{
    // in this example
    // pure vector is better
    int n = 10;
    vector<unique_ptr<int[]>> pascalTrig(n);

    for (int i = 0; i < n; i++)
    {
```

2.4 HEAP MEMORY ALLOCATION

C++ Alternative: Smart Pointers

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)
```

2.4 HEAP MEMORY ALLOCATION

C++ Alternative: Smart Pointers

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)
```

2.4 HEAP MEMORY ALLOCATION

C++ Alternative: Smart Pointers

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      }
```

2.4 HEAP MEMORY ALLOCATION

C++ Alternative: Smart Pointers

example/lec02/pascalTrig

```
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  }
```

2.5 CONST POINTERS

The fundamental purpose of `const`

2.5 CONST POINTERS

The fundamental purpose of `const`

- Protect data from being accidentally modified

2.5 CONST POINTERS

The fundamental purpose of `const`

- Protect data from being accidentally modified
- Improve code readability

2.5 CONST POINTERS

The fundamental purpose of `const`

- Protect data from being accidentally modified
- Improve code readability
- Compiler optimization opportunities

2.5 CONST POINTERS

The fundamental purpose of **`const`**

- Protect data from being accidentally modified
- Improve code readability
- Compiler optimization opportunities

```
const int size = 100;  // Constant declaration
```

2.5 CONST POINTERS

POINTER TO CONSTANT

```
const int* ptr;  // Pointer mutable, data immutable
int const* ptr1; // Same as ptr

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

- The pointer address can be modified
- Data cannot be modified via the pointer

2.5 CONST POINTERS

CONSTANT POINTER

```
int* const ptr = &var;   // Pointer immutable, data mutable
```

- The pointer address is fixed
- Data can be modified via the pointer

2.5 CONST POINTERS

CONSTANT POINTER TO CONSTANT

```
// Pointer and data are both immutable
const int* const ptr = &var;
```

2.5 CONST POINTERS

Form	Pointer Mutability	Data Mutability	Declaration Example
Regular Pointer	✔	✔	`int* ptr`
Pointer to Constant	✔	✘	`const int* ptr`
Constant Pointer	✘	✔	`int* const ptr`
Double-const Pointer	✘	✘	`const int* const ptr`

2.5 CONST POINTERS

```
// Correct: string literal is a constant
const char* str1 = "Hello";
// Error: prohibited since C++11 (requires forced conversion)
char* str2 = "World";
```

2.6 POINTERS AND FUNCTIONS

Functions can accept pointers as arguments and return pointers, but the following common mistakes should be avoided:

```
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 POINTERS AND FUNCTIONS

Function Pointers

```
// Function prototype declaration
int add(int, int);

// Function pointer declaration
int (*pf)(int, int) = &add;
int (*pf2)(int, int) = add; // also ok

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

2.6 POINTERS AND FUNCTIONS

Example: Newton's Method

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;
```

2.6 POINTERS AND FUNCTIONS

Example: Newton's Method

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;
```

2.6 POINTERS AND FUNCTIONS

Example: Newton's Method

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;
```

2.6 POINTERS AND FUNCTIONS

Example: Newton's Method

example/lec02/newton

```
 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)) /
17     return newton(func, guess - y / dy);
18 }
```

2.6 POINTERS AND FUNCTIONS

Example: Newton's Method

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)) /
17     return newton(func, guess - y / dy);
18 }
19
20 double equation1(double x) {
21     return x * x - 2;
22 }
23
```

2.6 POINTERS AND FUNCTIONS

Example: Newton's Method

example/lec02/newton

```
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      // using lambda function
33      std::cout << newton([](double x) {return x * x - 2;}) <<
34      return 0;
35  }
```

2.6 POINTERS AND FUNCTIONS

Example: Newton's Method

example/lec02/newton

```
16      double dy = (func(guess + delta) - func(guess - delta)) /
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;
```

2.6 POINTERS AND FUNCTIONS

Example: Newton's Method

example/lec02/newton

```
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      // using lambda function
33      std::cout << newton([](double x) {return x * x - 2;}) <<
34      return 0;
35  }
```

2.6 POINTERS AND FUNCTIONS

NOTE

Function pointers do not support arithmetic operations.

2.6 POINTERS AND FUNCTIONS

2.6 POINTERS AND FUNCTIONS

- Functions can still return function pointers, but their usage is rare in C++.

2.6 POINTERS AND FUNCTIONS

- Functions can still return function pointers, but their usage is rare in C++.
- Modern C++ can combine with lambda functions.

2.6 POINTERS AND FUNCTIONS

- Functions can still return function pointers, but their usage is rare in C++.
- Modern C++ can combine with lambda functions.
- In actual engineering, it is recommended to use `std::function` instead of direct function pointers.

2.7 STRING POINTERS

Basic declaration methods

```
const char* str1 = "Hello";   // Recommended: explicit constant
char str2[] = "World";        // Character array
char* str3 = str2;            // Pointer to array
```

2.7 STRING POINTERS

Memory Situation (Assuming first address is 0x1000)

Address	Data
0x1000	'H'
0x1001	'e'
0x1002	'l'
0x1003	'l'
0x1004	'o'
0x1005	'\0'

2.7 STRING POINTERS

Core features:

- Ends with null character `'\0'`
- String literals are stored in read-only data segments
- Pointer stores the address of the first character

2.7 STRING POINTERS

In C++, it is not recommended to use raw string pointers directly; instead, use `std::string` as an alternative.

2.8 COMMAND LINE ARGUMENTS

STANDARD PARAMETER FORMS

```
int main(int argc, char* argv[])  // Most commonly used form
int main(int argc, char** argv)   // Equivalent form
```

OTHER VALID FORMS

```
int main()      // Parameterless version
int main(void) // C-style explicit no parameters
```

2.8 COMMAND LINE ARGUMENTS

Parameter Meaning

Parameter	Name	Description
argc	Argument Count	Number of command line arguments (≥1)
argv	Argument Vector	Pointer to parameter string array

2.8 COMMAND LINE ARGUMENTS

Test: `example/lec02/args`

```cpp
#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 REFERENCE CONCEPT

1.1 BASIC DEFINITION

- **Alias Mechanism**: Creates a new name for an existing variable
- **Must Initialize**: Must be bound to a valid object at declaration
- **Cannot Rebind**: Cannot change the target after initialization

```
int x = 10;
int& ref = x;  // ref is an alias for x
ref = 20;      // Modifies the value of x
```

2.9 REFERENCE CONCEPT

CORE FEATURES

2.9 REFERENCE CONCEPT

CORE FEATURES

- **Type Safety**: Strictly matches the referenced type

2.9 REFERENCE CONCEPT

CORE FEATURES

- **Type Safety**: Strictly matches the referenced type
- **Automatic Dereferencing**: No special symbols needed when used

2.9 REFERENCE CONCEPT

CORE FEATURES

- **Type Safety**: Strictly matches the referenced type
- **Automatic Dereferencing**: No special symbols needed when used
- **Address Sharing**: Shares memory addresses with the referenced object

2.9 REFERENCE CONCEPT

Problems Solved by References

Pointer Problem	Reference Solution
Null pointer risk	Must initialize, cannot be null
Wild pointer issue	Cannot be changed after binding
Memory leak risk	Does not involve dynamic memory management
High syntax complexity	Automatic dereferencing, simple syntax
Double free risk	Naturally follows RAII[1] principles

1. RAII: Resource Acquisition Is Initialization, proposed by the father of C++ Bjarne Stroustrup

2.9 REFERENCE CONCEPT

Difference between References and Pointers

Feature	**Reference**	**Pointer**
Initialization Requirement	Must explicitly initialize	Can be delayed
Nullability	Cannot be null	Can be `nullptr`
Rebind	Cannot	Can change target
Memory Management	Not involved	Requires manual allocation/release
Syntax	Automatic dereferencing	Requires `*` and → operators

2.10 LEFT AND RIGHT VALUES

BASIC DEFINITION

Category	Characteristics	Example
Left Value	Persistent identity, addressable	Variable
Right Value	Temporary object, about to be destroyed	Literals, expression results

2.10 LEFT AND RIGHT VALUES

2.10 LEFT AND RIGHT VALUES

- Only left values can create references in the alias usage of references.

2.10 LEFT AND RIGHT VALUES

- Only left values can create references in the alias usage of references.
- Function parameters can add two & to represent right value references.

2.10 LEFT AND RIGHT VALUES

- Only left values can create references in the alias usage of references.
- Function parameters can add two **&** to represent right value references.

```
 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 LEFT AND RIGHT VALUES

- Only left values can create references in the alias usage of references.
- Function parameters can add two **&** to represent right value references.

```
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 LEFT AND RIGHT VALUES

- Only left values can create references in the alias usage of references.
- Function parameters can add two **&** to represent right value references.

```
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 REFERENCE

CORE FEATURES

- **Read-only Access**: Cannot modify the original object through the reference
- **Extend Lifetime**: Can bind to temporary objects
- **Compatibility**: Accepts both `const` and non-`const` objects

```
// Legal, extends the lifetime of the literal
const int& cref = 42;
```

TYPICAL APPLICATIONS OF REFERENCES

TYPICAL APPLICATIONS OF REFERENCES

- Function parameter passing (avoid copying large objects)

TYPICAL APPLICATIONS OF REFERENCES

- Function parameter passing (avoid copying large objects)
- Return protective access

TYPICAL APPLICATIONS OF REFERENCES

- Function parameter passing (avoid copying large objects)
- Return protective access
- Use with temporary objects

TYPICAL APPLICATIONS OF REFERENCES

Typical Mistake:

```
int& add(const int& a, const int& b)
{
    int c = a + b;
    return c; // Wrong! Return reference of a local variable
}
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (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;
}
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```
 1 #include <iostream>
 2 #include <utility>
 3 #include <optional>
 4 #include <cmath>
 5
 6 using namespace std;
 7
 8 using num = optional<double>;
 9
10 pair<num, num>
11 solve(double a, double b, double c)
12 {
13     if (a == 0)
14     {
15         if (b == 0)
16         {
17             return make_pair(nullopt, nullopt);
18         }
19         else
20         {
21             return make_pair(-c / b, nullopt);
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```cpp
#include <iostream>
#include <utility>
#include <optional>
#include <cmath>

using namespace std;

using num = optional<double>;

pair<num, num>
solve(double a, double b, double c)
{
    if (a == 0)
    {
        if (b == 0)
        {
            return make_pair(nullopt, nullopt);
        }
        else
        {
            return make_pair(-c / b, nullopt);
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```cpp
#include <iostream>
#include <utility>
#include <optional>
#include <cmath>

using namespace std;

using num = optional<double>;

pair<num, num>
solve(double a, double b, double c)
{
    if (a == 0)
    {
        if (b == 0)
        {
            return make_pair(nullopt, nullopt);
        }
        else
        {
            return make_pair(-c / b, nullopt);
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```cpp
#include <iostream>
#include <utility>
#include <optional>
#include <cmath>

using namespace std;

using num = optional<double>;

pair<num, num>
solve(double a, double b, double c)
{
    if (a == 0)
    {
        if (b == 0)
        {
            return make_pair(nullopt, nullopt);
        }
        else
        {
            return make_pair(-c / b, nullopt);
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```
 8 using num = optional<double>;
 9
10 pair<num, num>
11 solve(double a, double b, double c)
12 {
13     if (a == 0)
14     {
15         if (b == 0)
16         {
17             return make_pair(nullopt, nullopt);
18         }
19         else
20         {
21             return make_pair(-c / b, nullopt);
22         }
23     }
24     double delta = b * b - 4 * a * c;
25     if (delta < 0)
26     {
27         return make_pair(nullopt, nullopt);
28     }
21             return make_pair(-c / b, nullopt);
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```
18          }
19          else
20          {
21              return make_pair(-c / b, nullopt);
22          }
23      }
24      double delta = b * b - 4 * a * c;
25      if (delta < 0)
26      {
27          return make_pair(nullopt, nullopt);
28      }
29      double s = sqrt(delta);
30      double x1 = (-b + s) / (2 * a);
31      double x2 = (-b - s) / (2 * a);
32      return make_pair(x1, x2);
33  }
34
35  void print_result(const pair<num, num> &result)
36  {
37      if (result.first.has_value())
38      {
21              return make_pair(-c / b, nullopt);
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```
25      if (delta < 0)
26      {
27          return make_pair(nullopt, nullopt);
28      }
29      double s = sqrt(delta);
30      double x1 = (-b + s) / (2 * a);
31      double x2 = (-b - s) / (2 * a);
32      return make_pair(x1, x2);
33  }
34
35  void print_result(const pair<num, num> &result)
36  {
37      if (result.first.has_value())
38      {
39          if (result.second.has_value())
40          {
41              if (result.first.value() == result.second.value())
42              {
43                  cout << "There are two same results: "
44                       << result.first.value() << endl;
45              }
21              return make_pair(-c / b, nullopt);
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```
27          return make_pair(nullopt, nullopt);
28      }
29      double s = sqrt(delta);
30      double x1 = (-b + s) / (2 * a);
31      double x2 = (-b - s) / (2 * a);
32      return make_pair(x1, x2);
33  }
34
35  void print_result(const pair<num, num> &result)
36  {
37      if (result.first.has_value())
38      {
39          if (result.second.has_value())
40          {
41              if (result.first.value() == result.second.value())
42              {
43                  cout << "There are two same results: "
44                      << result.first.value() << endl;
45              }
46              else
47              {
21              return make_pair(-c / b, nullopt);
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```
37      if (result.first.has_value())
38      {
39          if (result.second.has_value())
40          {
41              if (result.first.value() == result.second.value())
42              {
43                  cout << "There are two same results: "
44                      << result.first.value() << endl;
45              }
46              else
47              {
48                  cout << "There are two results: "
49                      << result.first.value()
50                      << " and "
51                      << result.second.value()
52                      << endl;
53              }
54          }
55          else
56          {
21              return make_pair(-c / b, nullopt); "
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```
47              {
48                  cout << "There are two results: "
49                      << result.first.value()
50                      << " and "
51                      << result.second.value()
52                      << endl;
53              }
54          }
55          else
56          {
57              cout << "There is only one result: "
58                  << result.first.value() << endl;
59          }
60      } else {
61          cout << "There is no solution" << endl;
62      }
63  }
64
65  int main(int argc, char **argv)
66  {
67      auto r1 = solve(1, 2, 1);
21              return make_pair(-c / b, nullopt);
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```
51                      << result.second.value()
52                      << endl;
53              }
54          }
55          else
56          {
57              cout << "There is only one result: "
58                  << result.first.value() << endl;
59          }
60      } else {
61          cout << "There is no solution" << endl;
62      }
63  }
64
65  int main(int argc, char **argv)
66  {
67      auto r1 = solve(1, 2, 1);
68      auto r2 = solve(1, 1, 1);
69      auto r3 = solve(1, 1, -2);
70      auto r4 = solve(0, 1, 1);
71      auto r5 = solve(0, 0, 1);
21              return make_pair(-c / b, nullopt);
```

TYPICAL APPLICATIONS OF REFERENCES

Solving a quadratic equation (good way):

```
60      } else {
61          cout << "There is no solution" << endl;
62      }
63  }
64
65  int main(int argc, char **argv)
66  {
67      auto r1 = solve(1, 2, 1);
68      auto r2 = solve(1, 1, 1);
69      auto r3 = solve(1, 1, -2);
70      auto r4 = solve(0, 1, 1);
71      auto r5 = solve(0, 0, 1);
72
73      print_result(r1);
74      print_result(r2);
75      print_result(r3);
76      print_result(r4);
77      print_result(r5);
78      return 0;
79  }
21              return make_pair(-c / b, nullopt);
```