C/C++ Programming Language

CS205 Spring

Feng Zheng

2019.03.28

南方科技大学

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Content

- Brief Review
- Function Review
- Various Functions
- Summary

Brief Review

Content of Last Class

- Loops
  - for(; ;)
  - while( )
  - do while( )
  - Increment/decrement operations(++,--)
- Branching
  - if; if else; if else if else
  - switch
  - ?; continue; break;
- Loops
  - Relational expressions (6 operations)
  - Logical expressions (AND, OR, NOT)

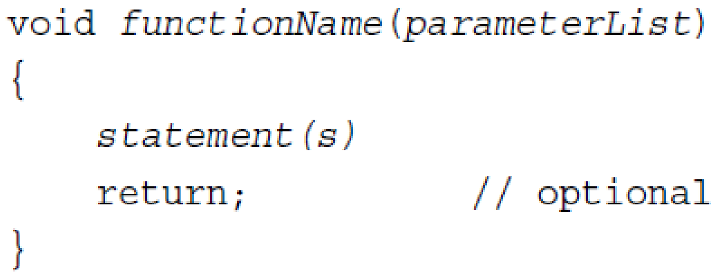
Function Review

Functions

- Three components
  - ➢ Provide a function definition
  - ➢ Provide a function prototype
  - ➢ Call the function
- Two types of usage
  - ➢ Use a library function
    - ✓ Including the header file
  - ➢ Create your own functions
    - ✓ Handle all three aspects

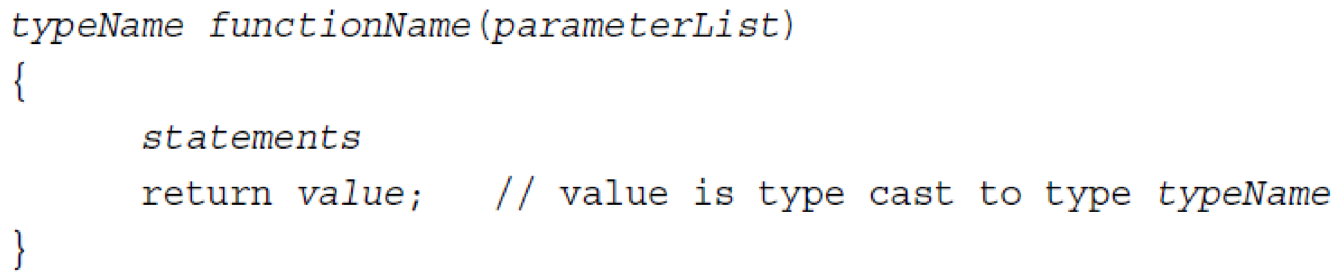
Defining a Function

- Two categories
  - ➢ Don't have return values



```
void functionName(parameterList)
{
      statement(s)
      return;            // optional
}
```

  - ➢ Do have return values
    - ✓Return value can be a constant, a variable, or a more general expression
    - ✓Both the returning function and the calling function have to agree on the type of data at that location
    - ✓The function terminates after it executes the first return statement it reaches



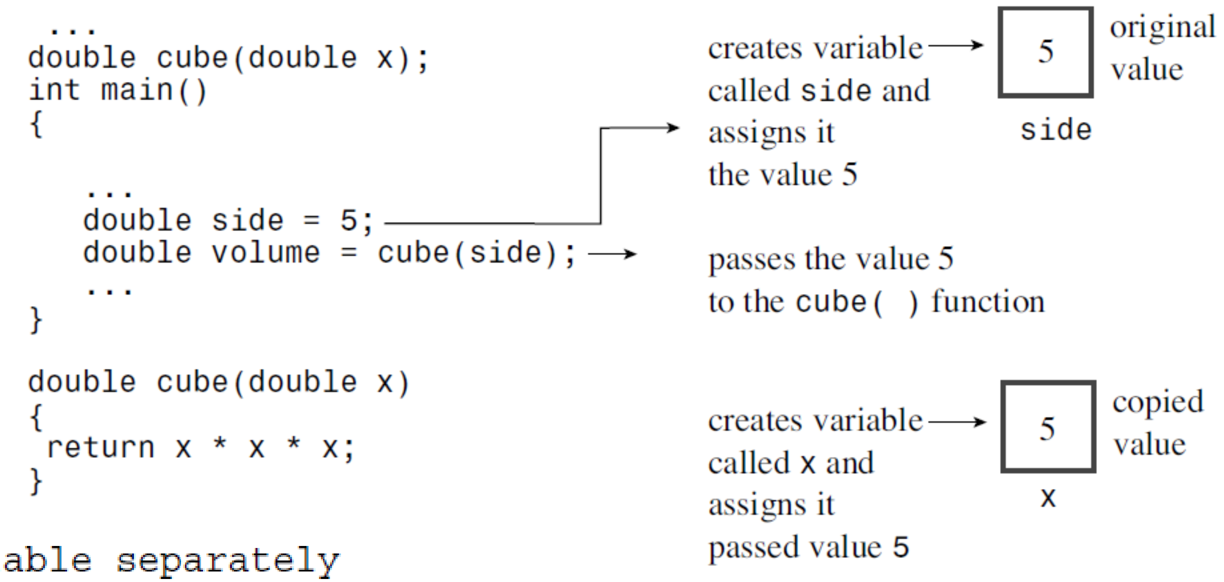
```
typeName functionName(parameterList)
{
      statements
      return value;   // value is type cast to type typeName
}
```

Prototyping and Calling a Function

- Why prototypes?
  - ➢ The function interface to the compiler
  - ➢ The only way to avoid using a function prototype is to place the function definition before its first use
  - ➢ Prototype syntax
    - ✓ A function prototype is a statement
    - ✓ Does not require that you provide names for the variables
- What prototypes do for you
  - ➢ The compiler handles the function return value
  - ➢ The compiler checks the number of function arguments
  - ➢ The compiler checks the type of arguments and converts the arguments to the correct type

Function Arguments and Passing by Value

- Call a function
  - Create a new type double variable--*formal argument or formal parameter*
  - Initialize it with the value--*actual argument* or *actual parameter*
  - Insulate data from the calling function--*rather than with the original data*
- Multiple Arguments
  - Have more than one argument
  - Comma is used
- See program example 1



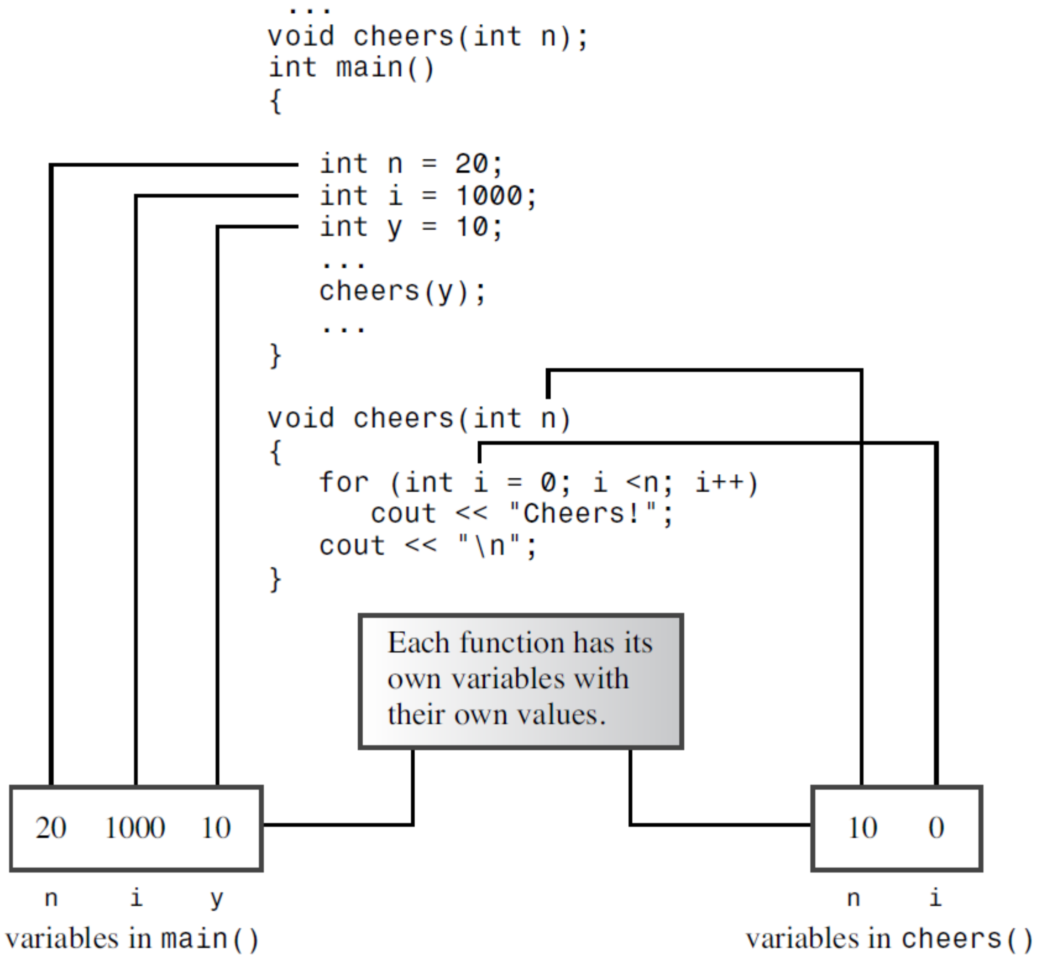
```
...
double cube(double x);
int main()
{
   ...
   double side = 5;
   double volume = cube(side);
   ...
}

double cube(double x)
{
 return x * x * x;
}
```

creates variable called side and assigns it the value 5
5
original value
side
passes the value 5 to the cube( ) function
creates variable called x and assigns it passed value 5
5
copied value
x

```
void fifi(float a, float b)   // declare each variable separately
void fufu(float a, b)    ⇦    // NOT acceptable
```

Local variables

- Automatic variables
  - Variables declared within a function are private to the function
  - They are allocated and deallocated automatically during program execution
  - When a function is called, the computer allocates the memory needed for these variables
  - When the function terminates, the computer frees the memory that was used for those variables



```
...
void cheers(int n);
int main()
{
    int n = 20;
    int i = 1000;
    int y = 10;
    ...
    cheers(y);
    ...
}

void cheers(int n)
{
    for (int i = 0; i <n; i++)
        cout << "Cheers!";
    cout << "\n";
}
```

Each function has its own variables with their own values.

Various Functions

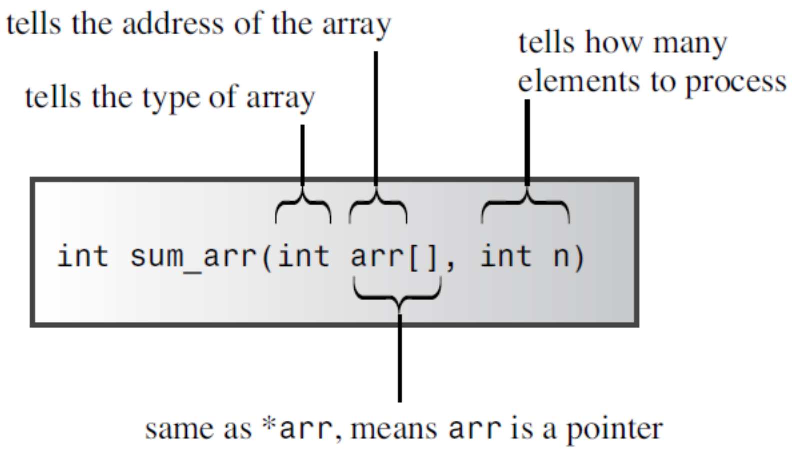
Functions and Arrays

- See program example 2
  - ➢ Suppose you use an array to keep track of how many cookies each person has eaten at a family picnic
- How pointers enable array-processing functions
  - ➢ Treat the name of an array as a pointer
  - ➢ There are a few exceptions to this rule
    - ✓ Use the array name to label the storage
    - ✓ sizeof operation yields the size of the whole array in bytes
    - ✓ Address operator & returns the address of the whole array
  - ➢ int *arr and int arr[]
    - ✓ Have the identical meaning when (and only when) used in a function header or function prototype
    - ✓ Not synonymous in any other context

More about Arrays for Functions

- The implications of using arrays as arguments
  - ➢ If you pass an ordinary variable, the function works with a copy.
  - ➢ If you pass an array, the function works with the original
  - ➢ Use array addresses as arguments saves the time and memory

- See program example 3
  - ➢ Explicitly pass the size of the array



tells the address of the array

tells the type of array

tells how many elements to process

```
int sum_arr(int arr[], int n)
```

same as *arr, means arr is a pointer

More Array Function Examples

- See program example 4
  - Fill the array
  - Show the array and protect it with const
  - Modify the Array
- Problems
  - Need to be informed about the kind of data in the array, the location of the beginning of the array, and the number of elements in the array
- See program example 5
  - Functions using array ranges

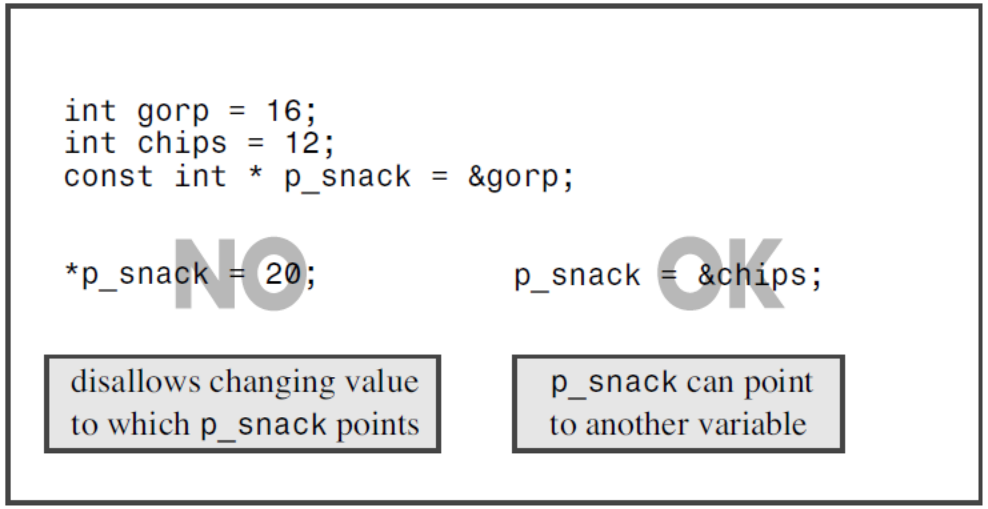
Pointers and const

- Make a pointer point to a constant object

```
int age = 39;
const int * pt = &age;

*pt += 1;          // INVALID because pt points to a const int
cin >> *pt;        // INVALID for the same reason

*pt = 20;         // INVALID because pt points to a const int
age = 20;         // VALID because age is not declared to be const


const float g_earth = 9.80;
const float * pe = &g_earth;   // VALID

const float g_moon = 1.63;
float * pm = &g_moon;          // INVALID
```

Pointers and const

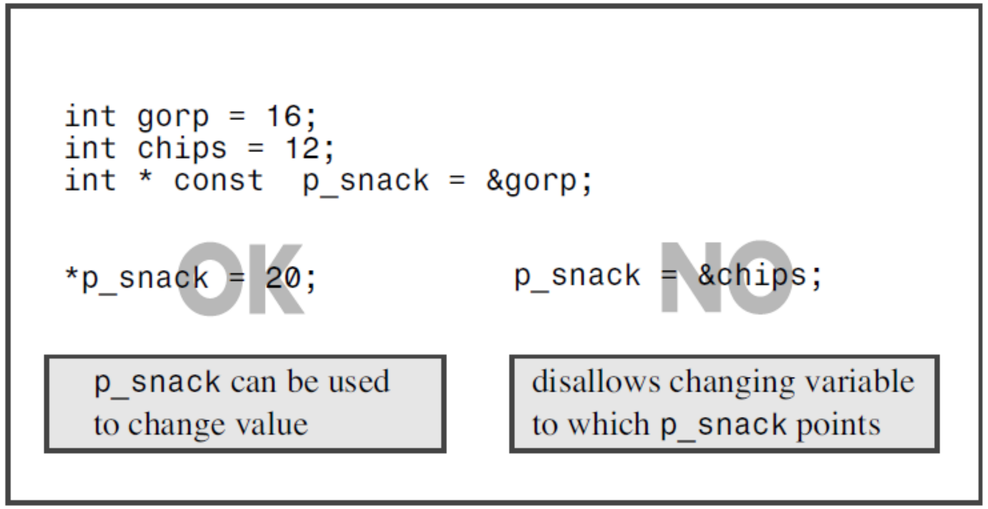
- Declare pointer arguments as pointers to constant data
  - It protects you against programming errors that inadvertently alter data
  - Using const allows a function to process both const and non-const actual arguments, whereas a function that omits const in the prototype can accept only nonconst data



```
int gorp = 16;
int chips = 12;
const int * p_snack = &gorp;

*p_snack = 20;                p_snack = &chips;
```

disallows changing value to which p_snack points

p_snack can point to another variable



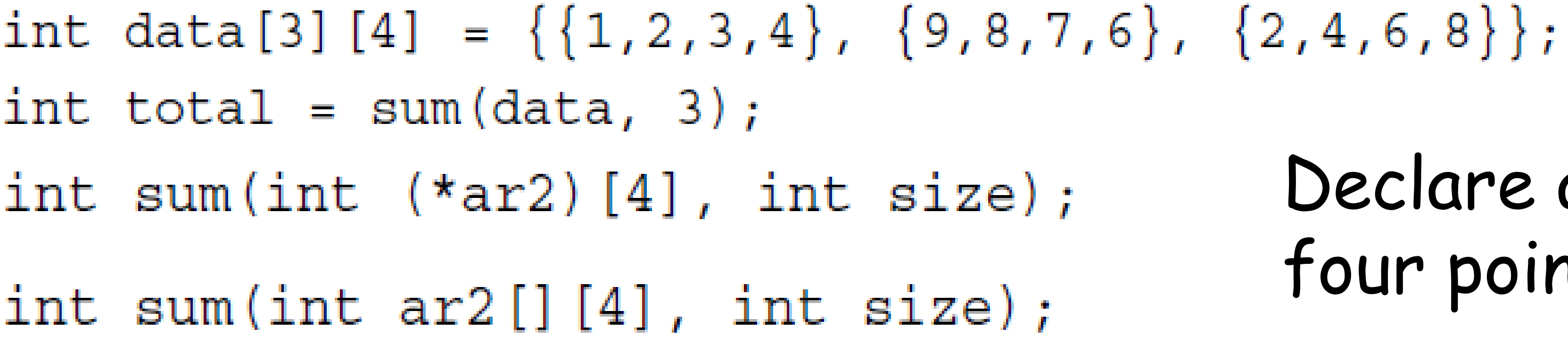
```
int gorp = 16;
int chips = 12;
int * const  p_snack = &gorp;

*p_snack = 20;                p_snack = &chips;
```

p_snack can be used to change value

disallows changing variable to which p_snack points

Functions and Two-Dimensional Arrays

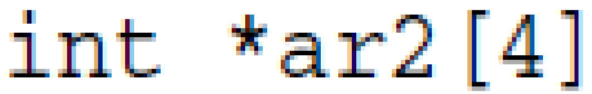
- The name of an array is treated as its address
  - ➢ The type of data is pointer-to-**array-of-four-int**



```
int data[3][4] = {{1,2,3,4}, {9,8,7,6}, {2,4,6,8}};
int total = sum(data, 3);
int sum(int (*ar2)[4], int size);

int sum(int ar2[][4], int size);
```

- See program example 6

Declare an array of four pointers-to-int



```
int *ar2[4]
```

Functions and C-Style Strings

- See program example 7
  - ➢ Functions with C-Style string arguments
    - ✓ An array of char
    - ✓ A quoted string constant (also called a string literal)
    - ✓ A pointer-to-char set to the address of a string
- See program example 8
  - ➢ Functions that return C-Style strings
  - ➢ It is not recommended to use new and delete separately

Functions and Structures

- A structure ties its data in to a single entity, or data object, that will be treated as a unit
  - A function can receive a structure
  - A function can return a structure
- Disadvantage
  - If the structure is large, the space and effort involved in making a copy of a structure can increase memory requirements and slow down the system
- See program example 9
  - Passing and returning structures

Passing Structure Addresses

- Save time and space
  - ➢ Pass it the address of the structure
  - ➢ Declare parameter to be a pointer-to- structure type
  - ➢ Use the indirect membership operator (->)

- See program example 10

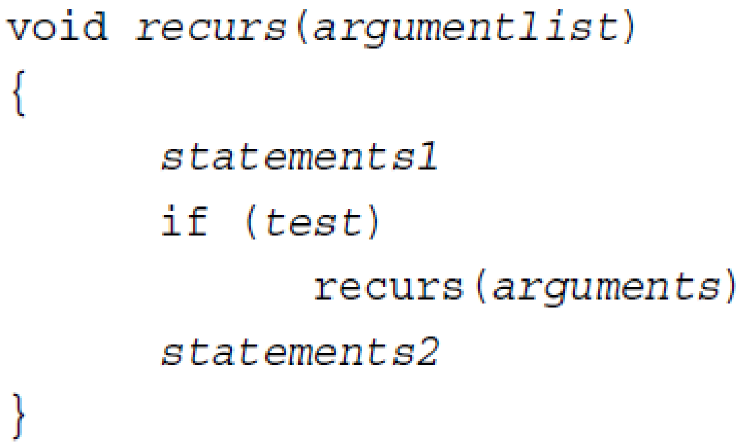
Functions and Two Class Objects

- Functions and string class objects
  - A string class object is more closely related to a structure than to an array
  - See program example 11

- Functions and array objects
  - See program example 12

Recursion

- C++ function has the characteristic that it can call itself
- C++ does not let main() call itself
  - See program example 13

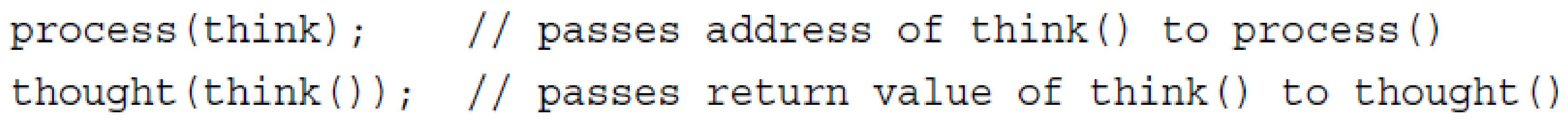


```
void recurs(argumentlist)
{
      statements1
      if (test)
            recurs(arguments)
      statements2
}
```

- Recursion with multiple recursive calls
  - Divide-and-conquer strategy (merge sort)
  - See program example 14

Pointers to Functions

- Functions, like data items, have addresses
  - ➢ The stored machine language code for the function begins
  - ➢ Write a function that takes the address of another function as an argument
- Three steps
  - ➢ 1: obtain the address of a function



```
process(think);     // passes address of think() to process()
thought(think());   // passes return value of think() to thought()
```

Pointers to Functions

➢ 2: declare a pointer to a function

```
double pam(int);  // prototype

double (*pf)(int);   // pf points to a function that takes
                     // one int argument and that
                     // returns type double
```

```
double (*pf)(int); // pf points to a function that returns double
double *pf(int);   // pf() a function that returns a pointer-to-double
```

➢ 3: use a pointer to invoke a function

```
double pam(int);
double (*pf)(int);
pf = pam;            // pf now points to the pam() function
double x = pam(4);   // call pam() using the function name
double y = (*pf)(5); // call pam() using the pointer pf
```

```
double ned(double);
int ted(int);
double (*pf)(int);
pf = ned;           // invalid -- mismatched signature
pf = ted;           // invalid -- mismatched return types
```

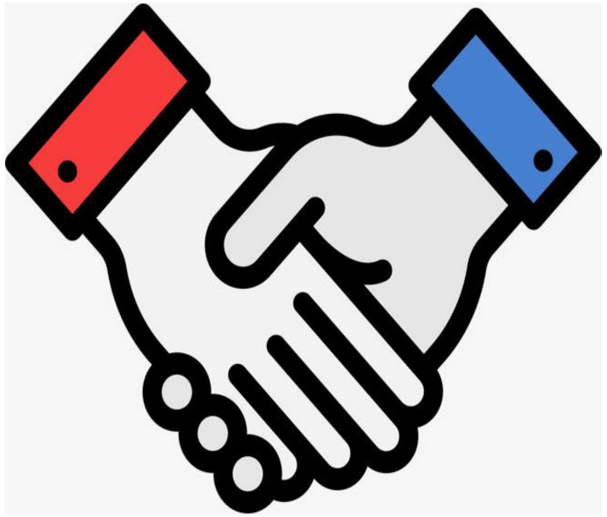
Two Function Pointer Examples

- See program example 15

- See program example 16
  - Variations on the theme of function pointers

Summary

- Function review
  - Function definition and prototype
  - Returned and passed values
  - Local values
- Various functions
  - Arrays
  - C-style
  - Structure
  - String class and array objects
  - Recursion
  - Pointer to functions



Thanks

zhengf@sustech.edu.cn