

C/C++ Programming Language

CS219 Fall

Feng Zheng

Lecture 6



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**;

- **The expressions for Loops**

- **Relational** expressions (6 operations)
- **Logical** expressions (**AND**, **OR**, **NOT**)



Function Review



Functions

- **Three** components
 - Provide a function **definition** including two parts: head and body
 - Provide a function **prototype**
 - **Call** the function
- **Two** types of usage
 - Use a **library** function
 - ✓ Including the header file
 - ✓ Static library is locked into at **compiling** time
 - ✓ Dynamic library exists as a **separate** file outside of the executable 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

- run `twoarg.cpp`

```
...  
double cube(double x);  
int main()  
{
```

```
    ...  
    double side = 5;  
    double volume = cube(side);  
    ...  
}
```

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

creates variable → 5 original value
called side and assigns it the value 5
side

passes the value 5 to the cube () function

creates variable → 5 copied value
called x and assigns it passed value 5
x

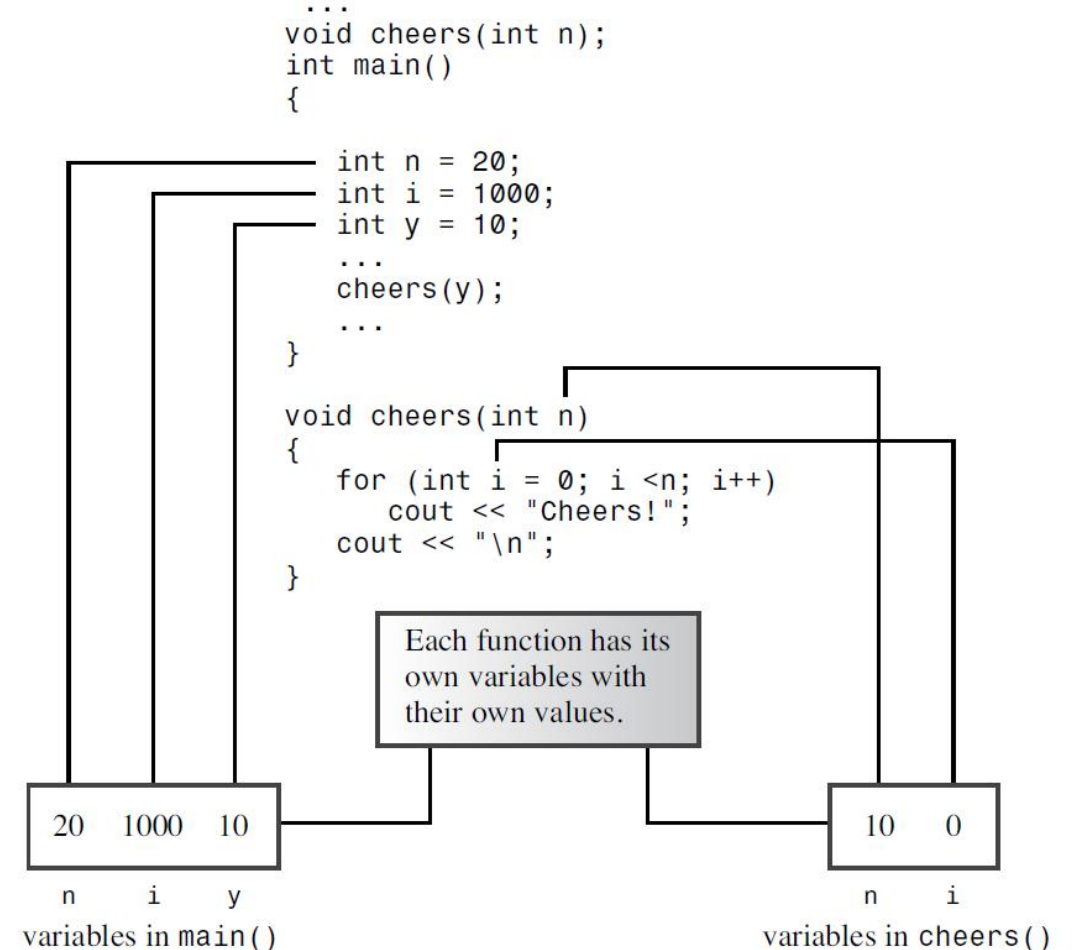
```
void fifi(float a, float b) // declare each variable separately
```

```
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 automatically **allocates** the memory needed for these variables
 - ✓ When the function **terminates**, the computer automatically **frees** the memory that was used for those variables



Various Functions



Functions and Arrays

- Run `arrfun1.cpp`

- 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

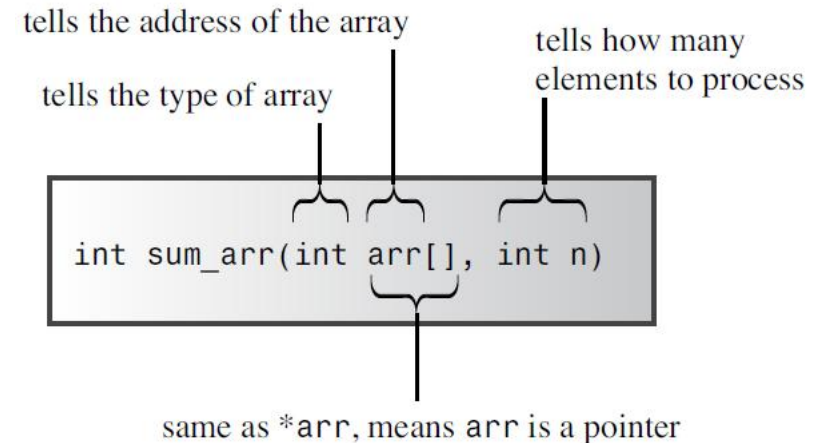
- Treat the **name** of an array as a **pointer**
- There are a few exceptions to this rule
 - ✓ Use the array name to label the storage
 - ✓ **sizeof** operation yields the size of the **whole array** in bytes
 - ✓ Address operator **&** returns the address of the **whole array**
- **`int *arr` and `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**

- Run **arrfun2.cpp**
 - Explicitly pass the **size** of the array





More Array Function Examples

- Run `arrfun3.cpp`

- 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

- Run `arrfun4.cpp`

- 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
```

The value stored in the location named as **age** cannot be altered by **pt**

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

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

```
int * const p1; // p1: read only *p1: variable
const int *p2; // p2: variable *p2: read only
int const *p3; // p3: variable *p3: read only
```



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;
```

NO
`*p_snack = 20;`

disallows changing value
to which p_snack points

OK
`p_snack = &chips;`

p_snack can point
to another variable

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

OK
`*p_snack = 20;`

p_snack can be used
to change value

NO
`p_snack = &chips;`

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);
```

- Run `sumArrayFun.cpp`

Declare an array of
four pointer-to-int



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



Functions and C-Style Strings

- Run `strgfun.cpp`

- 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

- Run `strgback.cpp`

- 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
- **Run travel.cpp**
 - Passing and returning structures

**How to check
whether is there a
copy?**



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 (->)
- Run `strctptr.cpp`



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**
 - Run `topfive.cpp`
- Functions and array objects (a type of **class**)
 - Run `arrobj.cpp`



Recursion

- C++ function has the characteristic that **it can call itself**

- Artificial intelligence

- C++ **does not** let **main()** call itself

- Run **recur.cpp**

- Recursion with **multiple recursive calls**

- Divide-and-conquer strategy (merge sort)

- Run **ruler.cpp**

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



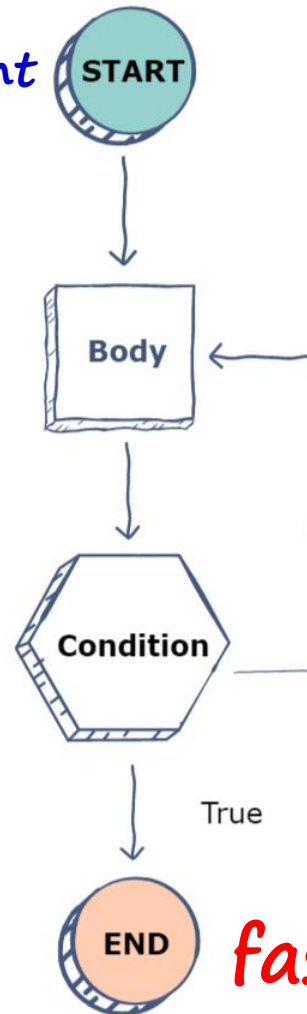
Difference Between Recursion and Iteration

Conditional control statement

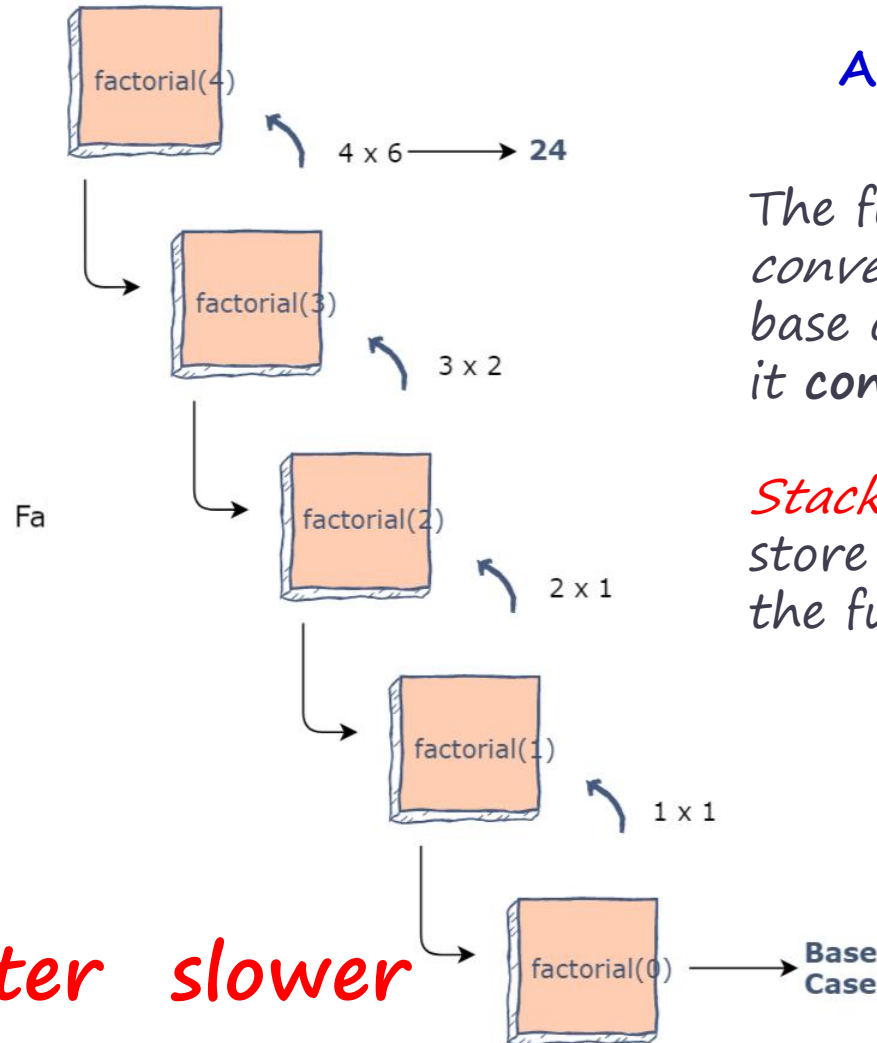
The value of the control variable continuously approaches the value in the conditional statement.

A control *variable* stores the value, which is then updated, monitored, and compared with the conditional statement.

Infinite loops keep utilizing CPU cycles until we stop their execution manually.



faster **slower**



At least one base case

The function keeps on converging to the defined base case as it continuously calls itself.

Stack memory is used to store the current state of the function.

If there is no base case defined, recursion causes a stack overflow error.



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
```

```
short tell[10];
short (*pas)[10] = &tell;
pas=&tell
```

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



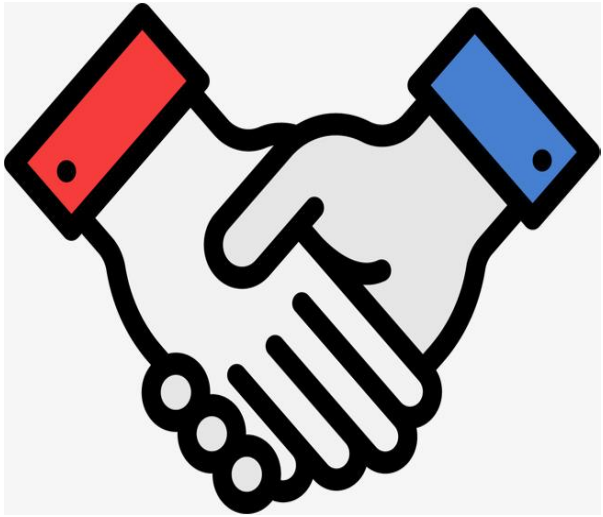
Two Function Pointer Examples

- Run `fun_ptr.cpp`
- Run `arfupt.cpp`
 - Variations on the theme of function pointers
 - `const double *(*(*pd)[3])(const double *, int) = &pa;`
 - ✓ `(*pd)` refers to the name of the array
 - ✓ The two parts, `const double *`; `(const double *, int)`, are the input and
 - ✓ output, respectively.
 - ✓ `*` refers to that the elements of the array are pointers-to-something.



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