Fundamentals of C++ Programming

Function and Pointer
Lecturer: Duc Dung Nguyen
Credits: 4
Outcomes

❖ Solving problems with functions
❖ Understand recursive algorithms
❖ Declare and implement recursive functions
❖ Declare and using pointers
Outline

❖ Function: definition, declaration, parameters, returned value
❖ Scope of variables
❖ Storage
❖ Pointer
❖ Recursion
Function

cuu duong than cong . com
Function

- You should never write **monolithic code**
  - Difficult to **write correctly**.
  - Difficult to **debug**.
  - Difficult to **extend**.
  - Hard to **maintenance**
  - Non-**reusable**
- **Nonsense!**
Math vs. Computer Software

- A function can return no value
- A function can take many different types of parameters
- A function can set as many output as it needs
Definition: a group of statements that is given a name, and which can be called from some point of the program.

Syntax:

- `<type> <name>(<parameters>);`
- `<type> <name>(<parameters>) { <statements> }`

- `<type>`: the value returned by the function
- `<name>`: name of function
- `<parameters>`: each parameter consists of a type followed by an identifier
Function

❖ `<type>`

❖ The function can return any type

❖ At some point, it must return a value

❖ `return x;`

❖ The function can return nothing (sometimes it is called procedure)

❖ No need for the `return` statement.

❖ `return` statement can be used to end the function.
Function

Example

```cpp
#include <iostream>
#include <math.h>

int generateArrayValue(int range) {
    return rand() % range;
}

int main() {
    int img[12][16];
    for (int i = 0; i < 12; i++) {
        for (int j = 0; j < 16; j++) {
            img[i][j] = generateArrayValue(256);
        }
    }
    return 0;
}
```
# Function

## Example

```cpp
#include <iostream>
#include <math.h>

float add(float a, float b) {
    return a + b;
}

int main() {
    float img[12][16];
    for (int i = 0; i < 12; i++) {
        for (int j = 0; j < 16; j++) {
            img[i][j] = add(rand() % 256, (rand() % 100) * 0.01);
        }
    }
    return 0;
}
```
#include <iostream>
#include <math.h>

void printArray(float*, int);
void printArray(float* pA, int N) {
    for (int i = 0; i < N; i++) {
        cout << pA[i] << " ";
    }
}

float add(float a, float b) {
    return a + b;
}

int main() {
    float img[12][16];
    for (int i = 0; i < 12; i++) {
        for (int j = 0; j < 16; j++) {
            img[i][j] = add(rand() % 256, (rand() % 100) * 0.01);
        }
    }
    printArray(img[2], 16); // print the third line of two dimensions array
    return 0;
}
Function

❖ Name
❖ Many functions can have the same name: overloaded functions.
❖ Functions with the same name must not share the same prototype
❖ Function signature: name + parameter list
❖ Provide convenience for programmer
# include <iostream>
# include <math.h>

float add(float a, float b) {
    return a + b;
}

int add(int a, int b) {
    return a + b;
}

double add(int a, double b) {
    return (double)a + b;
}

int main() {
    double k = 3.14159;
    cout << add(3.0, 5.2) << endl;
    cout << add(3, -8) << endl;
    cout << add(1, k) << endl;
    return 0;
}
Function

❖ **Parameters**: there are two ways to pass parameters to a function

❖ **Value**: the value will be copied to local variable (parameter) of the function

❖ **Reference** (only in C++): the parameter is associated with passed variable

❖ User can only pass variables through a reference parameter

❖ Any change in the parameter affects the variable
Function

Example

```cpp
#include <iostream>
#include <math.h>

float add(float a, float b) {
    b += 1;
    return a + b - 1;
}

float foo(int a, float &b) {
    b *= a;
    return b;
}

int main() {
    float x = 2.5, y = 3.14159;
    cout << add(3.1, 6.08) << endl;
    x = foo(2, y);
    cout << x << endl;
    cout << y << endl;
    return 0;
}
```
**Function**

- **main:**
  - Default return value of main: 0 - the program executed successfully
- **stdlib.h/cstdlib:**
  - **EXIT_SUCCESS:** same as default return value
  - **EXIT_FAILURE:** the program failed
Function

❖ Parameter passing:
  ❖ C++ allows user pass parameters by value or by reference
  ❖ If user pass a parameter using reference, it will be translated to pointer
  ❖ Unlike C++, everything in Java is pass-by-value.
  ❖ Think about what happens in the background.
  ❖ C++ allows user pass default values to parameters
Function

Example

```cpp
#include <iostream>
#include <math.h>

float add(float a, float b = 1.0f) {
    return a + b;
}

int main() {
    float x = 2.5;
    cout << "add: x + 6.08 = " << add(x, 6.08) << endl;
    cout << "increase x: x + 1 = " << add(x) << endl;
    return 0;
}
```
Function

- Reuse functions
  - Define prototype in header file (.h): `<type> <name>(<parameters>);`
  - Must export the function if it was build in a library.
    - Use export/import instructions: depend on platform and language
- Static linked libraries vs. Dynamic linked libraries
Function

❖ Why do you need function prototype?
❖ To reuse a function written in another module
❖ To solve tricky situations
Function

- **inline functions**
  - Similar to function, except that the compiled code will be inserted where we call inline functions.
  - Purpose: improve performance
  - **inline** `<return type> <function name>(<parameters>) {<function body>}

  [CuuDuongThanCong.com](https://fb.com/tailieudientucntt)
Scope of Variables
Scope of Variables

• In C/C++, the variable is effective in the scope of declaration statement

• In C: all variables must be declared at the beginning of the function. No initialization in declaration.

• In C++: variables can be declared anywhere and take effect in the declared scope

```c
void test() {
    for (int i = 0; i < 5;)
    {
        i += 2;
    }

    i = 10; // error
}
```
Scope of Variables

❖ Global vs. Local variables

❖ Global variables can be accessed everywhere in the function without declaration

❖ Local variables can only be accessed inside the scope where it is declared
Scope of Variables

❖ Global vs. Local variables

```c++
#include <iostream>
#include <math.h>

float defaultFactor;

float mul(float a, float b, bool useGlobal = false) {
    return useGlobal? a * defaultFactor: a * b;
}

int main() {
    defaultFactor = 2.0f;
    cout << "Use default factor: " << mul(3.14159, 0, true) << endl;
    cout << "Multiply pi by 5: " << mul(3.14159, 5.0f) << endl;
    return 0;
}
```
Scope of Variables

❖ Global vs. Local variables
❖ Why don’t we declared everything at global scope?
❖ Benefit of local variable?
❖ When should we use global variables?
❖ When should we use local variables?
Scope of Variables

❖ Global vs. Local variables
❖ Local variables take precedence over global variables
❖ The :: operator is called the scope resolution operator
Scope of Variables

❖ Global vs. Local variables

```cpp
#include <iostream>
#include <math.h>

float accSum;

float acc(float a, float accSum) {
    ::accSum += a;
    return accSum + a;
}

int main() {
    accSum = 0.0f;
    cout << "acc(3.14159, 0): " << acc(3.14159, 0) << endl;
    cout << "acc(3.14159, 1): " << acc(3.14159, 1) << endl;
    cout << "accSum: " << accSum << endl;
    return 0;
}
```
Storage
Storage

❖ How your program is organized?
❖ What are common errors?
❖ Memory overflow
❖ Memory corruption
Storage

high address

command line arguments and environment variables

/ / / /

stack

cuu duong than cong . com

heap

uninitialized data (bss)

cuu duong than cong . com

initialized data

read from program file

low address

initialised to zero by exec

text (code segment)
Storage

✦ Code segment: contains executable code (binary code)
✦ Data segment:
  ✦ Initialized data: global, static, constants
  ✦ Uninitialized data
✦ Heap: contains allocated memory at runtime
✦ Stack: stores local variables, passed arguments, and return address
Storage

❖ Common errors:
❖ Use variables without initialization
❖ Memory fault
❖ Access restricted areas
❖ Overwrite meta information on memory
❖ Stack overflow
Storage

❖ Uninitialized variables

#include <iostream>
#include <math.h>

float __gVal;  

float foo(float a, float b) {
  __gVal += b;
  return a * b + __gVal;
}

int main() {
  float x, y;
  x = 0.5f;
  cout << foo(x, y) << endl;
  return 0;
}
Storage

❖ Memory fault (access freed memory)

```cpp
#include <iostream>
#include <math.h>

float* foo(float a, float b) {
    a += b;
    return &a;
}

int main() {
    float x, y;
    x = 0.5f;
    y = 3.9f;
    float *pRet = foo(x, y);
    cout << *pRet << endl;
    return 0;
}
```
Storage

- Memory fault (access restricted area)

```cpp
#include <iostream>
#include <math.h>

char* getConstString() {
    return "This is a string";
}

int main() {
    char* pStr = getConstString();
    cout << pStr << endl;
    for (int i = 0; i < 10; i++) {
        pStr[i] = '-';
    }
    cout << pStr << endl;
    return 0;
}
```
Storage

- Overwrite meta information in memory (memory corruption)

```c
#include <iostream>
#include <math.h>

void foo(char *pStr) {
    char buf[10];
    strcpy(buf, pStr);
}

int main() {
    char* pStr = "This string will overwrite the local buffer";
    foo(pStr);
    return 0;
}
```
❖ Blow away your stack (stack overflow)

```cpp
#include <iostream>
#include <math.h>

int foo(int n) {
    return n + foo(n + 1);
}

int main() {
    cout << "This code will blow away your stack\n";
    foo(0);
    return 0;
}
```
Storage

❖ How to avoid memory errors? Heap vs. Stack errors
❖ Invalid memory access
❖ Memory leaks
❖ Mismatched allocation/deallocation
❖ Missing allocation
❖ Uninitialized memory access
❖ Cross stack access
MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?

I HATE YOU.
What is pointer?

How do we access memory so far?

Through variables: use identifiers

What if you need something more dynamic?

Allocate on demand

Vary in size

Flexible access mechanism
C++ program does not decide exact memory address of variables, the OS does.

Obtain memory address of a variable

Use operator &

E.g.:
float a = 8.534f;
cout << &a << endl;
Declare a pointer variable

\[
\text{<type>} \; * \; \text{<identifier>};
\]

E.g.:

\[
\text{int} \; * \; \text{pInt}, \; a;
\]

Define a pointer type

\[
\text{typedef <type>}* \; \text{<alias\_type>};
\]

E.g.: \textbf{typedef int} \; * \; \text{intPointer};
Pointer

- Special values
  - Specific areas managed by OS
  - Stack
  - Code address
  - Data addresses
- NULL == 0 (== nullptr)
#include <iostream>
#include <math.h>

int main() {
    float x = 3.14159;
    float* pX;
    pX = &x;
    cout << "Value of x: " << x << endl;
    cout << "Address of x: " << pX << endl;
    *pX *= 2.5;
    cout << "Value of x: " << x << endl;
    pX = NULL;
    return 0;
}
Pointer

- Dereference operator *
  - Used to access the memory pointed to by the pointer variable
  - Usage: *<pointer variable>
  - Example:
    ```
    int *pX = &a;
    *pX = 5;
    a = *pX - 2;
    ```
Casting pointer value

```cpp
int *p;
p = 0xff63; // error, illegal statement
```

```cpp
int *p;
p = reinterpret_cast<int *>(0xff63);
```

Use pointer wisely!
Pointer

- Pointer vs. array (static)
  - Pointer can be changed!
  - Access memory in the same way
  - A static array can be considered as a constant pointer

```cpp
#include <iostream>
#include <math.h>

int main() {
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = 0;
        cout << a[i] << " ";
    }
    cout << endl;
    int *pA = a;
    for (int i = 0; i < 10; i++) {
        pA[i] = i + 1;
        cout << pA[i] << " ";
    }
    cout << endl;
    *a = i;
    cout << " ";
    return 0;
}
```
Passing array as function parameter

- Passing values: define a new structure to hold array

  ```
  int foo(int a[10]) {
      ...
  }

  int foo(int a[]) {
      ...
  }
  ```

```cpp
#include <iostream>
#include <math.h>

typedef struct {
    int data[10];
} myStruct;

int foo(myStruct a) {
    int sum = 0;
    for (int i = 0; i < 10; i++)
        sum += a.data[i];
    return sum;
}

int main() {
    myStruct mA;
    for (int i = 0; i < 10; i++) {
        mA.data[i] = i;
    }
    cout << "Sum = " << foo(mA) << endl;
    return 0;
}
```
Passing array as function parameter

Passing with pointers

```cpp
#include <iostream>
#include <math.h>

int foo(int *a) {
    int sum = 0;
    for (int i = 0; i < 10; i++)
        sum += a[i]++;
    return sum;
}

int main() {
    int iVals[10];
    for (int i = 0; i < 10; i++)
        iVals[i] = i;
    cout << "Sum = " << foo(iVals) << endl;
    for (int i = 0; i < 10; i++) {
        cout << iVals[i] << " ";
    }
    return 0;
}
```
Pointer

- Pointer vs. Reference
  - Reference is a convenient definition for pointer
    - E.g.:
      ```
      int &rv = v;
      int *pv = &v;
      rv = 10;
      *pv -= 10;
      ```
  - Pointer can be changed, but Reference
 POINTER

❖ Pointer vs. const pointer

❖ Pointer: access memory address with read/write privileges

❖ Const pointer: only read memory address

❖ E.g.:
  ```
  int x, y = 10;
  const int * p0 = &y; // pointer to a constant
  int * const p1 = &y; // a constant pointer
  x = *p0; // OK
  *p0 = x; // error
  p1 = &x; // error
  ```
Casting problem

```c
int * pointer = "Demon";
int * const const_pointer0;
int * const const_pointer1 = pointer;
const int * pointer_to_const;

pointer = const_pointer; // OK, no cast
pointer_to_const = pointer; // OK, casting from (int*) to (const int*)
pointer = pointer_to_const; // Illegal
const_pointer1 = "Evil";
```
**Pointer**

❖ **Pointer to pointer**

❖ **E.g.:**

```c
int x = 9;
int *pX = &x;
int **pPx = &pX;
**pPx = 8;
*pPx++;
```

❖ **pPx++**;
Pointer

- Void pointer
  - An special type of pointer representing the absent of type
  - Declaration: `void * <pointer>;
  - Casting operations are necessary
  - Other operations:
  
  ```
  p++; ++p;  p--; --p;
  p += N;  p -= N;
  ```
Pointer

- Void pointer
  - Great flexibility: can point to any data type
  - Used in adaptive structures
  - Used in function definitions
Order of operators

- \( *p++ \) // \( *(p++) \)
- \( +++p \) // \( *(++p) \)
- \( +++p \) // \( +(*(p)) \)
- \( (*p)++ \) // increase value at location pointed by \( p \)
  // (only valid with integer pointers)

When in doubt, use safe statement.
**Pointer**

- Allocate memory
  - C: malloc/free
  - C++: new/delete
    - `int *pN = new int;`  
    `delete pN;`
    - `float *pF = new float[N];`  
    `delete [] pF;`
**Pointer**

- Allocate memory
  - Dynamic memory: there are cases where memory needs of a program can only be determined in runtime.

- Common mistake:
  - Forget release memory! (don’t be a Java programmer)
  - Overwrite heap information by accessing outside the allocated area
  - Overflow
**Function pointer**

A pointer can be used to call a function

<return type> (*<pointer name>)(<parameter type>);

E.g.:

```c
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
...
int (*op)(int, int);
op = add;
cout << (*op)(1, 2);
```
Function pointer

Define function pointer type:

- `typedef <return type> (*<type name>)(<parameter type>);`

Easy to read and organise code

E.g.:

```c
typedef int (*usrOp)(int, int);
usrOp myFuncPointer;
if (usrSel == 0) myFuncPointer = add;
else myFuncPointer = sub;
```
Recursion
Recursion

- Problem solving methods
  - Principle: divide the big problem into smaller problems
- Recursivity is a property that function have to be called by themselves.
  - Principle: define the solution of big problem using the solution of smaller problems. A set of base solution must be defined
- E.g.: Fibonacci sequence is defined as follows
  - \( F(n) = F(n - 1) + F(n - 2) \)
  - \( F(1) = F(2) = 1 \)
Recursion

❖ E.g.:
❖ Factorial function: \( f(n) = n! \)
❖ By definition:
   ❖ \( 0! = 1 \)
   ❖ \( f(n) = f(n - 1) \times n \)
❖ Solution: Iterative algorithm vs. Recursive algorithm
❖ Fibonacci function vs. Factorial function
Recursion

❖ More examples
❖ Simple: print a string backward
❖ Classic: Hanoi tower
Recursion

❖ Type of recursions

❖ Tail recursion: nothing has to be done after the call return
❖ Head recursion: the first statement in function is a recursive call
❖ Middle / multi-recursion
❖ Mutual recursion: function X and Y are mutually-recursive if function X calls function Y, and function Y in turn call function X. This is called indirect recursion
“Never hire a developer who computes the factorial using Recursion.”

– Unknown recruiter
Recursion

- Recursive solution is often more elegant and easier to spot than iterative solutions
- E.g.: Hanoi tower problem
- Iterative solutions are usually more efficient than recursive solutions
Recursion

- Recursive method vs. Iterative method
  - Are equally expensive: why?
  - Where should you use recursive?
- Functional languages: Lisp, Scheme, Haskell, Erlang, F#, D, R, Scala, etc.
- Overhead:
  - Imperative language: define function
  - Functional language: define accumulator variable
Can you remove recursion in function? (Refactoring)

Yes

Based on recursive mechanism

- Stack structure
- Return points
Extras

❖ How do you define `printf`, `scanf`?

❖ Function with variable number of arguments

❖ Standard Args (require `stdarg.h`)
  ❖ `<type> <function>(<parameters>, ...);`
  ❖ `va_list args;` // declare arguments list
  ❖ `va_start(args, var);` // initialise argument list, pop first argument
  ❖ `var = va_arg(args, <type>);` // pop next argument
  ❖ `va_end(args);` // clear argument list
Extras

❖ Function vs. inline functions
❖ Macros
❖ Template
Summarise

❖ Learn about functions: how to define and use in the program
❖ How to pass parameters, understand scope of variables
❖ Memory organization of a program
❖ Pointer
❖ Recursion technique
Quiz & homework