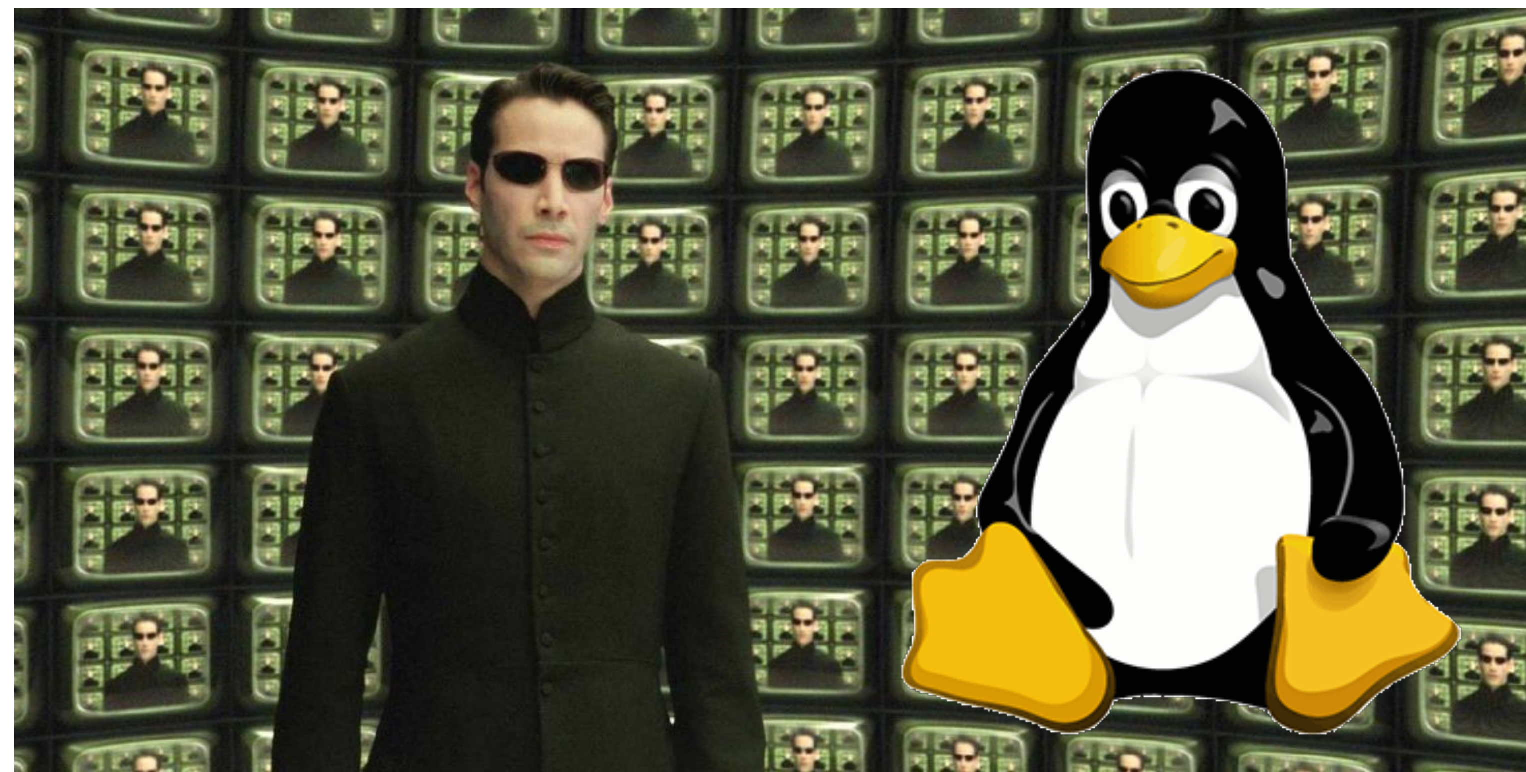
18.06.20 19:33

# CSCI 125
# Introduction to Computer Science and Programming II
# Lecture 4: Function II



Jetic Gū

2020 Summer Semester (S2)

# Overview

- Focus: Basic C/C++ Syntax
- Architecture: Linux/Unix OS
- Core Ideas:
  1. Value and Reference Parameters
  2. Const and Static Variables
  3. Overloading

P0
Review

# Data Types

- Primary √
    - Integers, Characters, Boolean √
    - Floating point √
    - Void √
- Derived
    - Function, Array√, Pointer√, Reference√
- User Defined
    - Struct, Class, Enumerate, Typedef

Review

# Value and Reference Parameters

Summary

# Function Parameters

- Value parameter: a formal parameter that receives a copy of the content of corresponding actual parameter
    - Including pointer, in which contents are addresses
- Reference parameter: a formal parameter that receives the location (memory address) of the corresponding actual parameter
    - Value passing through aliasing

Concept

# Value Parameters

- If a formal parameter is a value parameter
    - The value of the corresponding actual parameter is **copied** into it
- **The value parameter has its own copy of the data**
- During program execution
    - The value parameter manipulates the data stored in its own memory space
    - e.g. `pow(2, 10);`

Concept

P2
Pointer

# Value Parameters

- Value Parameter
  - Parameter value copied

1. ...
2. ...
3. Allocate memory for **`a`**
4. Enter subroutine
   allocate memory for **`n`**
   init with value of **`a`**

```
1. int foo(int n);
2. ...
3. int a=10;
4. foo(a);
```

| Address | Values |
|---|---|
| 0x0004: a | 10 |
| 0x0008: foo/n | 10 |

Technical

P1
Parameters

# Reference Parameters

- If a formal parameter is a reference parameter
  - It receives the memory address of the corresponding actual parameter
- A reference parameter stores the address of the corresponding actual parameter
- During program execution to manipulate data
  - The address stored in the reference parameter directs it to the memory space of the corresponding actual parameter

Concept

P2
Pointer

# Reference Parameters

- Reference Parameter
  - Parameter used as alias

1. ...
2. ...
3. Allocate memory for **a**
4. Enter subroutine
   variable **n** and **a** are the **same** in this **scope**

```
1. int foo(int& n);
2. ...
3. int a=10;
4. foo(a);
```

| Address | Values |
|---|---|
| 0x0004: a, foo/n | 10 |
| 0x0008 | |

Technical

# Reference Parameters

- Reference parameters can:
  - Pass **one or more** values from a function
  - **Change the value** of the actual parameter
- Reference parameters are useful in three situations:
  - Returning more than one value
  - **Changing the actual parameter**
  - **When passing the address would save memory space and time**

P2 Pointer

# Swapping Function (Reference)

- Reference Parameter
  - Parameter used as alias

1. ...
2. ...
3. ...
4. Allocate memory for **`x, y`**
5. Enter subroutine
   variable **`a`** and **`x`** are the **same** in this scope, **`b`** and **`y`** are the **same**

```
1. int swap(int& a, int& b) {
2.       int tmp=a; a=b; b=tmp;
3. }
4. int x=10, y=5;
5. swap(x, y);
```

| Address | Values |
| --- | --- |
| 0x0004: x, foo/a | 5 |
| 0x0008: y, foo/b | 10 |

Example

# Swapping Function (Pointer)

- Variable Parameter
  - Parameter value copied

1. ...
2. ...
3. ...
4. Allocate memory for **x, y**
5. Enter subroutine
   allocate memory for **a, b**

```
1. int swap(int* a, int* b) {
2.      int tmp=*a; *a=*b; *b=tmp;
3. }
4. int x=10, y=5;
5. swap(&x, &y);
```

| Address | Values |
| --- | --- |
| 0x0004: x | 5 |
| 0x0008: y | 10 |
| 0x000c: foo/a | 0x0004 |
| 0x0010: foo/b | 0x0008 |

Example

# Value and Reference Parameters and Memory Allocation

- When a function is called
    - Memory for its formal parameters and variables declared in the body of the function (called local variables) is allocated in the function data area
- In the case of a value parameter
    - The value of the actual parameter is copied into the memory cell of its corresponding formal parameter

Concept

P1
Parameters

# Value and Reference Parameters and Memory Allocation

- In the case of a reference parameter
  - The address of the actual parameter passes to the formal parameter
- Content of formal parameter is an address
- During execution, changes made by the formal parameter permanently change the value of the actual parameter

Concept

P1
Parameters

# Reference Parameters and Value-Returning Functions

- You can also use reference parameters in a value-returning function
  - Not recommended
- By definition, a value-returning function returns a single value
  - This value is returned via the return statement
- If a function needs to return more than one value, you should change it to a void function and use the appropriate reference parameters to return the values

Concept

# Const and Static Variables

More variable types!

Summary

P2
Static Variables

# Automatic Variables

- **Automatic variable**: memory is allocated at subroutine/block entry and deallocated at subroutine/block exit
    - Standard variables
    - Pointer variables: the memory space for **storing the address**
      ! actual space for value storage is managed **manually**
    - Reference variables
    - Parameters of Functions

Review

# `const` Variables

- Syntax

```
const DataType variableName=InitialValue;
```

  - e.g.
```
const int n=5;
```

- Require initialisation

- Once declared, its value will not change, this is ensured by the compiler
  The compiler checks the address you are trying to modify and see if it’s "protected by `const`"

Concept

# const Variables

- This will not compile
  `error: cannot assign to variable 'n' with const-qualified type 'const int'`

```
#include <iostream>
using namespace std;
int main() {
    const int n=10;
    n = 5;
    cout << n << endl;
    return 0;
}
```

P2
Static Variables

# `const` Variables

- This does compile
- But does it work?

```
#include <iostream>
using namespace std;
int main() {
    const int n=10;
    int* m;
    m = (int*)&n;
    *m += 5;
    cout << n << endl;
    return 0;
}
```

Technical

# Static and Automatic Variables

- **Automatic variable**: memory is allocated at subroutine entry and deallocated at block exit
  - Any variables declared within a block are automatic variables
- **Static variable**: memory remains allocated as long as the program executes/library included
  - Variables declared outside of any block are static variables
  - Declare a static variable within a block by using the reserved word `static`

Concept

# Static and Automatic Variables

- Syntax

```
static DataType variableName;
```

  - e.g.

```
static int x;
// x is a static variable of the type int
```

- Static variables declared within a block are local to the block
  - Their scope is the same as any other local identifier of that block

Concept

P2
Static Variables

# Static Variables

- When will variable `num` be allocated memory?
- When will variable `num` be deallocated? (recycled)
- What will the output be?
- What is the scope of variable `num`?

```
#include <iostream>
using namespace std;
void func() {
    static int num = 1;
    cout <<"Value of num: "<< num <<"\n";
    num++;
}
int main() {
    func();
    func();
    func();
    return 0;
}
```

Technical

# Function Overloading, Default Parameters

Summary

# Introduction

- **Function overloading:** several functions can have the **same name identifier**
- Requirement: have **different formal parameter lists**
  - Different **number** of formal parameters; or
  - Different data **type**(s)
  - Parameter names **doesn't** matter
  - Return type **doesn't** matter

Concept

P3 Overloading

# Overloading Requirement

- The following functions all have **different formal parameter lists**, can be overloaded

```
void f(int x);
void f(int x, double y);
void f(double y, int x);
int f(char ch, int x, double y);
int f(char ch, int x, string y);
```

- The following functions have the **same formal parameter list**, no overloading:
  - `void f2(int x, double y, char ch);`
  - `void f3(int one, double two, char three);`

Concept

# Why Overloading?

- Custom **op** that can work with all kinds of data types

```
int      op(int x,     int y);
string   op(string x,  string y);
MyClass  op(MyClass x, MyClass y);
```

- Functions need to specify types, this way you can easily reuse your code
  - May not appear as intuitive, but very fast

Concept

P3
Overloading

# Functions with Default Parameters

- In a function call, the number of actual and formal parameters must be the same
  - C++ relaxes this condition for functions with default parameters
- You specify the value of a default parameter when the function name appears for the first time (e.g., in the prototype)
- If you do not specify the value of a default parameter, the default value is used

Concept

P3
Overloading

# Functions with Default Parameters

- All default parameters must be the rightmost parameters of the function
- In a function call where the function has more than one default parameter and a value to a default parameter is not specified:
  - You must omit all of the arguments to its right
- Default values can be constants, global variables, or function calls
  - However, you cannot assign a constant value as a default value to a reference parameter

Concept

P3
Overloading

# Functions with Default Parameters

- Consider the following prototype:

```
void funcExp(int x, int y, double t, char z = 'A',
             int u=67, char v='G', double w=78.34);
int a, b; char ch; double d;
```

- Examples of legal calls:

```
funcExp(a, b, d);
funcExp(a, 15, 34.6, 'B', 87, ch);
funcExp(b, a, 14.56, 'D');
```

- Examples of illegal calls:

```
funcExp(a, 15, 34.6, 46.7, 87, ch);
funcExp(b, 25, 48.76, 'D', 4567, 78.34);
```

Concept

P3
Overloading

# Functions with Default Parameters

- Examples of illegal function prototypes with default parameters:

```
int    funcExp(int x, double z=11.12, char ch, int u=45);
double funcExp(int length=1, int width, int height=1);
void   funcExp(int x, int& y=16, double z=34);
```

Concept