EE538: Computing Principles for Electrical Engineers

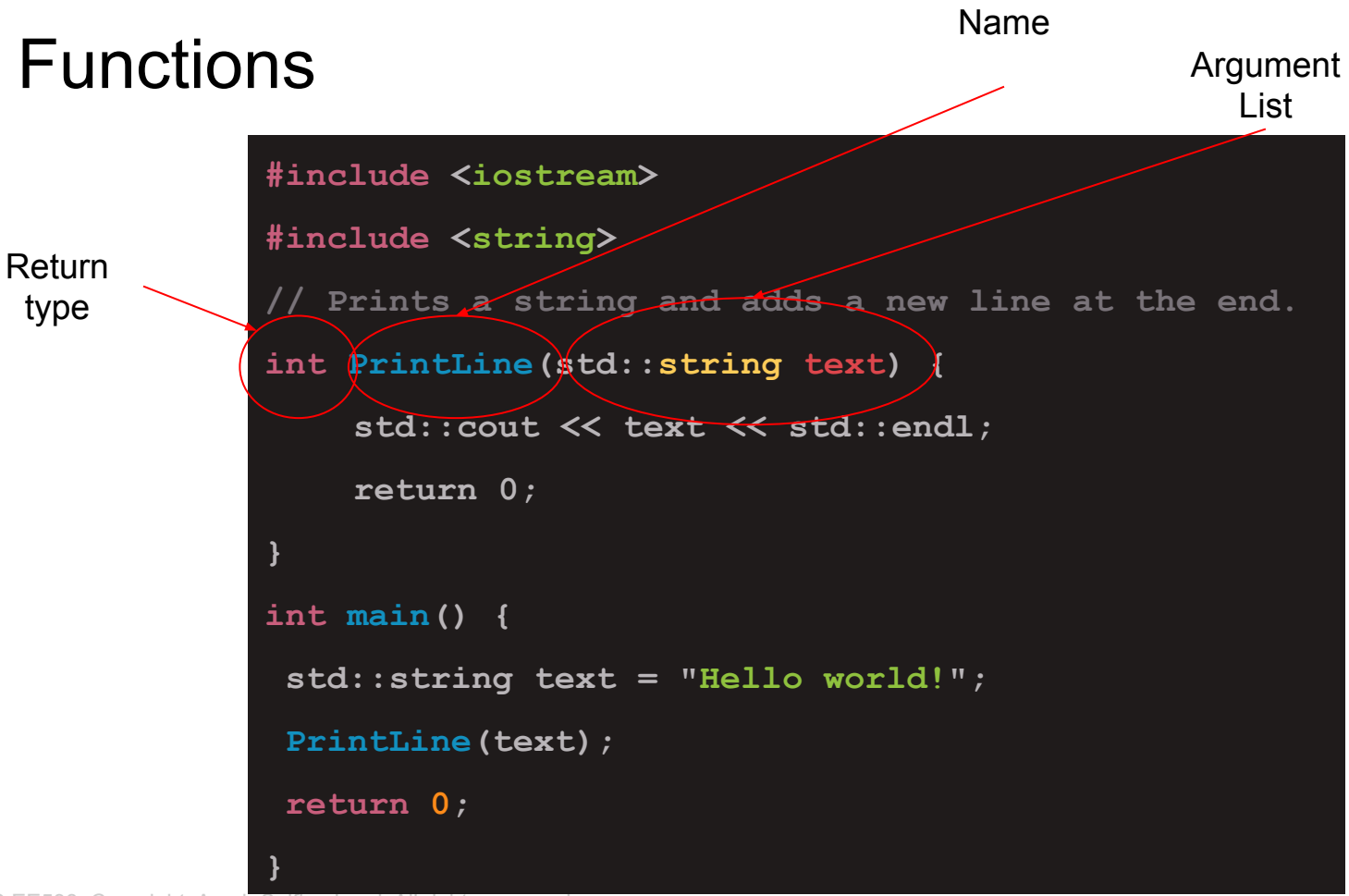
Discussion 3: C++ basics, Reference & Pointer

University of Southern California
06/01,06/02 Summer 2022
Instructor: Arash Saifhashemi
TA and Mentors:Yijun Liu, Aditi Bodhankar, Zixu Wang

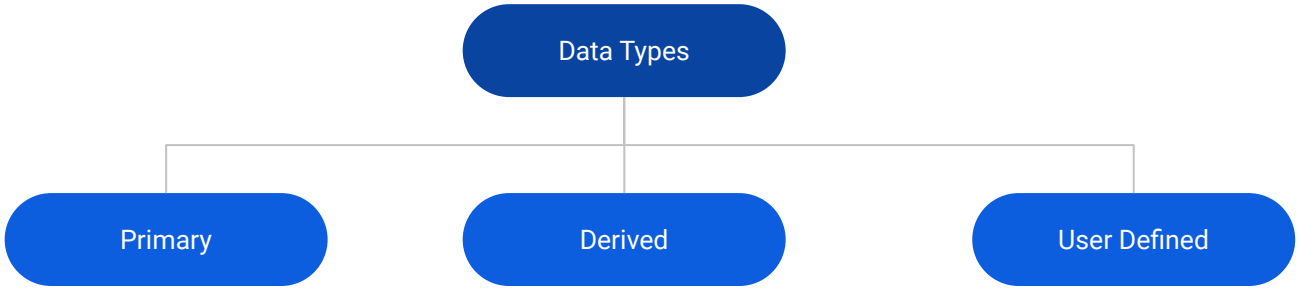
Outline

- C++ basics
  - Functions
  - Variables
- Reference
- Pointer
- Array
- Dynamic Memory Allocation
- Pass by ref vs Pass by pointer

Functions



Data Types



- `int`
- `char`
- `bool`
- `float`
- `double`
- `void`

- `array`
- `pointer`
- `references`

- `struct`
- `class`
- `union`
- `enum`
- `typedef`

Variables

```
int main() {

 std::string hello = "Hello";
 std::string world = " world";
 int year = 2020;

 std::string hello_world = hello + world +
     " " + std::to_string(year);

 PrintLine(hello_world);
 return 0;
}
```

Enum Class

- Scoped enumeration
  - Strongly typed
  - Strongly scoped
- Use **enum class** instead of just **enum**!

```
// Enum type in C:
enum ColorPallet1 { Red, Green, Blue };
enum ColorPallet2 { Yellow, Orange, Red };

// Enum Class in C++
// Declaration
enum class ColorPalletClass1 { Red, Green, Blue };
enum class ColorPalletClass2 { Yellow, Orange, Red };

// Assignment
ColorPalletClass1 col1 = ColorPalletClass1::Red;
ColorPalletClass2 col2 = ColorPalletClass2::Red;
```

auto

```
int main() {
 std::vector<int> my_vector = {1, 2, 3, 4, 5, 6, 7, 8};
 for (auto n : my_vector) {
   std::cout << n << std::endl;
 }
  return 0;
}
```

I'm too lazy to specify the type,
please figure it out yourself!

Pointer

- A pointer is a variable that holds memory address of another variable. A pointer needs to be dereferenced with * operator to access the memory location it points to.

Pointer

Rockman
int x = 10;

Spiderman
int* ptr = &x;

How to find Rockman?

1. Directly call Rockman
   Use x
2. Ask Spiderman, since Spiderman knows where is Rockman
   *Use *ptr*
   *(ptr store the address of x, and * is the dereference operation that could access the address!)*

Pointer initialization

```
int x = 0;
// Pointer
int* y = &x;
```

```
int x = 0;
// Pointer
int* y = nullptr;
y = &x;
```

Pointer example

```
int x = 0;
// With Pointer
int* ptr = &x;
std::cout << &x << " " << ptr << std::endl;
std::cout << &ptr << std::endl;
std::cout << x << " " << *ptr <<std::endl;
*ptr = 10;
std::cout << x << " " << *ptr <<std::endl;
x = 20;
std::cout << x << " " << *ptr <<std::endl;
```

- *ptr* is a pointer point to *x.*
- **ptr* is as same as *x.*
- Modifying **ptr* will change the value of *x.*
- Modifying *x* will change the value of **ptr.*

Pointer example

```
int x = 0;
// With Pointer
int* ptr = &x;
std::cout << &x << " " << ptr << std::endl;
std::cout << &ptr << std::endl;
std::cout << x << " " << *ptr <<std::endl;
*ptr = 10;
std::cout << x << " " << *ptr <<std::endl;
x = 20;
std::cout << x << " " << *ptr <<std::endl;
```

Will the address of x and the value of ptr same or different?

Will the address of x and the address of ptr same or different?

What is the expected output?

What if we change y to 10?

What if we change x to 20?

Pointer example

```
int x = 0;
// With Pointer
int* ptr = &x;
std::cout << &x << " " << ptr << std::endl;
std::cout << &ptr << std::endl;
std::cout << x << " " << *ptr <<std::endl;
*ptr = 10;
std::cout << x << " " << *ptr <<std::endl;
x = 20;
std::cout << x << " " << *ptr <<std::endl;
```

```
0x7436dc196e04 0x7436dc196e04
0x7436dc196e08
0 0
10 10
20 20
```

Pointer usages - Pass by pointer

```
void swap (int* first, int* second)
{
    int temp = *first;
    *first = *second;
    *second = temp;
}
```

```
int main()
{
    int a = 2, b = 3;
    std::cout << a << " " << b << std::endl;
    swap(&a, &b);
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

Array

- Different Initialization:
  - Initialize array with zeros by default, then assign values to certain indices:

```
int arr[8];

arr[0] = 5;

arr[2] = -10;
```

  - Initialize array with specific values:

```
int arr[5] = {16, 2, 77, 40, 12071};
```

  - More conveniently:

```
int arr[5] {16, 2, 77, 40, 12071};
```

- No need to call **delete** after using, because it's on the stack.

Dynamic Memory Allocation

```
int main()
{
    // Below variables are allocated memory dynamically on heap.
    int *ptr1 = new int;
    int *ptr2 = new int[10];

    // Dynamically allocated memory is deallocated
    delete ptr1;
    delete [] ptr2;
}
```

1 **delete** call for every **new**. If you have no **new**, you don't **delete**.

Pointer usages - Dynamic array

```
int* arr;
arr = new int[3];
for(int i = 0; i < 3; i++){
    arr[i] = 0;
}
for(int i = 0; i < 3; i++){
    std::cout << arr[i] << " ";
}
std::cout << std::endl;
for(int i = 0; i < 3; i++){
    *(arr+i) = i*i;
}
for(int i = 0; i < 3; i++){
    std::cout << *(arr+i) << " ";
}
delete [] arr;
```

Pointer usages - Data structure

```
 struct TreeNode {
        int val;
       TreeNode *left;
       TreeNode *right;
}
```

```
class Node {
    public:
        int val;
        Node* next;
};
```

Advantages of Pointer

- Variables can passed without copy, which is fast! (Reference also do that!)
- Pointer could be pointer to nullptr!
- Pointer could be updated, which means it could point to other variable! (Reference is unable to do that!)
- (Advanced) We could have multiple indirection! Ex: int *** ptr;




...


Reference vs Pointer

- Use references whenever you can, and pointers when you have to.
- Pointer has more freedom; however it is harder to use and easily make mistakes!
- Reference:
  - In function parameters and return types
- Pointer:
  - Use pointers if pointer arithmetic or passing NULL pointer is needed. For example, arrays!
  - To implement data structures like linked list, tree, etc and their algorithms because to point different cell, we have to use the concept of pointers.

Practice (5 minutes)

Write 2 functions that will swap the values of the inputs by pointers and references.

- pass by pointers

```
void SwapByPointer(double *input1,double *input2);
```

- pass by references

```
void SwapByReference(double &input1, double &input2);
```

Example :

Before: x = 9.9, y = 7.5

Call Swap(x,y)

After: x = 7.5, y = 9.9

http://cpp.sh/

Solution - Pass by pointer

```
void swap (double* first, double* second)
{
    double temp = *first;
    *first = *second;
    *second = temp;
}
```

```
int main()
{
    double a = 2, b = 3;
    std::cout << a << " " << b << std::endl;
    swap(&a, &b);
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

Solution - Pass by reference

```
void swap (double& first, double& second)
{
    double temp = first;
    first = second;
    second = temp;
}
```

```
int main()
{
    double a = 2, b = 3;
    std::cout << a << " " << b << std::endl;
    swap(a, b);
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

Practice (15 minutes)

Given two integer vector `nums1` and `nums2`, return *an vector of their intersection*. Each element in the result must appear as many times as it shows in both arrays and you may return the result in any order.

```
vector<int> intersect(vector<int>& nums1, vector<int>& nums2)
```

Example 1:

```
Input: nums1 = [1,2,2,1], nums2 = [2,2]
Output: [2,2]
```

Example 2:

```
Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
Output: [4,9]
Explanation: [9,4] is also accepted.
```

Please do it by yourselves first before looking at solutions
http://cpp.sh/

Practice (15 minutes)

Given two integer vectors `nums1` and `nums2`, return a vector of their intersection. Each element in the result must appear as many times as it shows in both vectors and you may return the result in any order.

`vector<int> intersect(vector<int>& nums1, vector<int>& nums2)`

Hint:

Please use std::unordered_map and std::vector!

A linear complexity solution is preferred!

Solutions

```
vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
    unordered_map<int, int> dict;
    vector<int> res;
    for(int i = 0; i < nums1.size(); i++)
    {
        dict[nums1[i]]++;
    }
    for(int i = 0; i < nums2.size(); i++)
    {
        if(dict[nums2[i]] > 0)
            res.push_back(nums2[i]);
            dict[nums2[i]]--;
    }
    return res;
}
```

or

```
for(auto digit : nums1)
{
    dict[digit]++;
}
for(auto digit : nums2)
{
    if(dict[digit] > 0)
        res.push_back(digit);
        dict[digit]--;
}
```

Solutions

```
vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
    unordered_map<int, int> dict;
    vector<int> res;
    for(int i = 0; i < nums1.size(); i++)
    {
        dict[nums1[i]]++;
    }
    for(int i = 0; i < nums2.size(); i++)
    {
        if(dict[nums2[i]] > 0)
            res.push_back(nums2[i]);
            dict[nums2[i]]--;
    }
    return res;
}
```

Complexity: O(n1+n2)