RC2

Before the RC

- Focus on the content of slides , finish project and exercise by yourself.
- The slides will help you do well in the exam and homework will help you gain coding skills.
- I have found some of you using generative-LLM in the homework and project...

Const Qualifier

- Constant is defined as a global variable usually
  - Cannot be modified later and must be initialized when it is defined.
- Constant improves readability.

```
//
const int a=10;
a=11;//Error
const int b; // Error
```

Const Reference

Reference is an alias (alternative name) for an object. Once bound to the object, it cannot be changed to refer to another object. But the object can be modified through the reference.
We have properties:

- Reference must be initialized with an object.
- Const reference can be initialized to an rvalue.
- In contrast, nonconst reference cannot be initialized to an rvalue(must be initialized with an lvalue).

```
//Difference between ref and const ref
const int& cRef = 5; //OK
int& ref = 5; // ERROR
```

Usage of reference in function call

```
//
struct VeryBigStruct {
    double data[10000];
    double moreData[10000];
    double evenMoreData[10000];
};
void modify(VeryBigStruct& vbs); // Question: why do not use const here?
//Answer:Pass-by-value can be expensive, particularly for large structures.

void print(const VeryBigStruct& vbs); // Question: Why we use const reference here?
//Answer: If we dont use "const" here, It allows for the possibility of (mistakenly) changing th
```

Advantages of using const reference as argument

- We don't have the expense of a **copy**, copy is expensive for big data structure.
- We have the safety guarantee that the function cannot change the caller's state.

Another example in the slides

```
//
foo("Hello world!")
void foo(string & str) {...} //error
void foo(const string &str) {...}// OK
```

Const pointer

Consider the three following types.

```
//
const int* ptr2Intc; // the integer cannot be modified
int const* ptr2cInt;// the integer cannot be modified
int* const cPtr2Int;// the address cannot be modified
```

How to remember: read backwards

For example

```
//
const int* A; // A is pointer to int const
int const* B;// B is pointer to const int(int const is same as const int )
int* const C;// C is const pointer to int
```

- For const pointer, you cannot modify the address it point to.
- For pointer to const, you cannot modify the value which the address store.

For example

- For a const pointer ptr, you cannot write: ptr=&x
- For a pointer to const ptr, you cannot write: *ptr=5

Now, look into the code provided below. If I compile it, will it be successful or not?

```
//
const int cInt = 0;
int anotherInt = 1;
int* ptr1 = &cInt; //errpr
const int *ptr2 = &cInt;
int* const ptr3 = &cInt; // error
int* const ptr4;
const int* ptr5;
ptr1 = &anotherInt;
ptr2 = &anotherInt;
ptr3 = &anotherInt
// For a int, you can transform it to const int!!
// For a const int, you cannot transform it to int!!
```

Why we use const pointer?

Same reason as const reference, we use const pointer to make sure some data will not be changed.

- Pointers-to-const-T are not the same type as pointers-to-T.
- You can use a pointer-to-T anywhere you expect a pointer-to-const-T, but NOT vice versa.

```
int const_ptr(const int *ptr)
{
    …
}

int main()
{
    int a = 0;
    int *b = &a;
    const_ptr(b);
}
```

```
int nonconst_ptr(int *ptr)
{
    …
}

int main()
{
    int a = 0;
    const int *b = &a;
    nonconst_ptr(b);
}
```

Typedef

typedef gives an alias to an existing type

```
//
typedef existingType newName;
typedef const int const_int;
typedef const_t* ptr_const_int;
```

See the fun quiz mentioned in the slides.

How do we use `typedef` to rename the type of `T *const`? const pointer

Select all the correct answers.

- **A.**
```
typedef const T const_t;
typedef const_t * constptrT_t;
```
- **B.**
```
typedef T * ptrT_t;
typedef const ptrT_t constptrT_t;
```
- **C.**
```
typedef const * constptr_t;
typedef constptr_t T constptrT_t;
```
- **D.**
```
typedef T * const constptrT_t;
```

12

For A, pointer to const T.
For B, pointer to T is const.
For C, Not valid.
For D, const pointer to T.

We expect T * const, which is const pointer. So we choose B, D.

Procedural Abstraction

Why Abstraction? Provide only details that matter.

Abstraction tell you "what to do" and Implementation tell you "how to do". Users only need to know what a function does, instead of knowing how it does.
The feature is quite useful in collaboration and maintainance.

For example

```
int swap(int& a, int& b);
 // REQUIRES: none
 // MODIFIES: x, y
 // EFFECTS: swaps the values of x and y

int factorial(int n);
  // REQUIRES: n >= 0
 // EFFECTS: returns n!
```

Given above information, clients can use the function without caring about how it is implemented. The implementation can be modified without affecting function calls.

Recursion

Find the subproblem and base case. Call itself with the subproblem until the base case.

```
  // REQUIRES: n >= 0
  // EFFECTS: returns n!
int factorial(int n) {
    if (n == 0) {
        return 1;  // base case
    } else {
        return n * factorial(n- 1); // recursive step
    }
  }
```

Recursive helper function

Recursive functions themselves should be able to be called with a subproblem.

If not, use a helper function.

```
//
bool isPalindrome(const string& s);
bool isPalindrome_helper(const string& s, int start, int end);
```

Function Pointer

Functions can now be referred to as variables, passed as arguments, etc. Consider two recursive functions to find the largest/smallest element in an array, respectively. The only difference is the

comparison.

```
//
int largest(int *arr, int length)
{
    if (length == 1)return arr[0];
    int first = arr[0];
    int *rest = arr + 1;
    int candidate = largest(rest, length - 1);

    return (first > candidate) ? first : candidate;
}
int smallest(int *arr, int length)
{
    if (length == 1)return arr[0];
    int first = arr[0];
    int *rest = arr + 1;
    int candidate = largest(rest, length - 1);
    return (first < candidate) ? first : candidate;
}
```

So, instead of writing two functions, we can write a single function that handles both cases.

```
int find(int* arr, int length, bool (*compare)(int, int)) {
    if (length == 1) return arr[0];
    int first = arr[0];
    int* rest = arr + 1;
    int candidate = find(rest, length- 1, compare);
    return compare(first, candidate) ? first : candidate;
}
```

Where compare is a function (pointer) that takes two int and returns a bool. The following two functions can be passed as arguments because they have the same signature.

```
//
bool larger(int a, int b) { return a > b; }
bool smaller(int a, int b) { return a < b; }
```

Solve the problem by yourself

- Given an array of integers, return the sum.

- Given an array of integers, return the product.
- Use function pointer to avoid writing the “same” code twice.

Function Call Mechanism

The function call are stored in stack, which follows the principle first in last out.

Stack
"First in Last out" principle, supporting following instruction.

- push, insert a element to the top of the stack
- pop, remove the element from the top of the stack
- peak, get the element of the stack
- isEmpty, check whether the stack is empty.

Think about how to implement them by array, Linked list.

Queue
"First in First out" principle, supporting following instruction.

- push, insert a element to the tail of the queue
- pop, remove the element from the front of the queue
- front, get the element of the front
- isEmpty, check whether the queue is empty.

Think about how to implement them by array, Linked list.

Enum

You can define an enumeration type as follows:

```
//
enum Suit_t {CLUBS, DIAMONDS,
HEARTS, SPADES};
```

Then we have numerically

```
//
CLUBS = 0, DIAMONDS = 1,
HEARTS = 2, SPADES = 3;
```

It will make life easier in the following code block.

```
//
Suit_t s = CLUBS;
const string suitname[] = {"clubs",
"diamonds", "hearts", "spades"};
std::cout << "suit s is " << suitname[s]; // directly use s as the index.
```

It may be used in the project 4 if the material is not changed.

Refernece

- ECE2800J-25FA Lecture Slides
- ECE2800J-25SP Lecture Slides
- ECE2800J-25SP RC2