



# VE280 Mid RC Part 1

---

## Before We Start

---

In big RC, I won't spend much time discussing the detailed motivations of some concepts. Instead, I will focus on the key points and the things that you might encounter in the exam. If you want to know more about the motivations and details, you can refer to the slides or my regular RC.

### Some tips for the exam

- BE HONEST
- Be careful
- Be critical

## Lecture 2: Linux Commands

---

### Basic Commands

- `man <command>` : read the help page of a command
- `pwd` : print working directory
- `ls` : list files and directories
  - `ls -a` : list all files and directories including hidden ones
  - `ls -l` : list files and directories in long format

Pay attention to the permissions of long format of `ls` :

- The first character indicates the type of the file
  - `-` : regular file
  - `d` : directory
- The next 9 characters indicate the permissions of the file
  - `r` : read
  - `w` : write

- `x` : execute
- The first three characters indicate the permissions of the owner
- The second three characters are for the group
- The last three characters are for others

◦ Example:

```
-rwxr-xr-- 1 user group 0 Jan 1 00:00 file
```

- `cd <path>` : change to the directory specified by `<path>`

Pay attention to the following special directories:

- `.` : current directory
- `..` : parent directory
- `~` : home directory
- `/` : root directory
- `mkdir <directory>` : make a new directory specified by `<directory>`
- `rmdir <directory>` : remove an empty directory specified by `<directory>`
- `touch <file>` : create an empty file specified by `<file>`
- `rm <file>` : remove a file or directory specified by `<file>`
  - `rm -r <file>` : remove a directory recursively

**Necessary for removing non-empty folders**

  - `rm -i <file>` : ask for confirmation before removing a file or directory
- `cp <source> <destination>` : copy a file or directory from `<source>` to `<destination>`
  - `cp -r <source> <destination>` : copy a directory recursively

**Necessary for copying non-empty folders**
- `mv` : move a file or directory or rename a file
  - `mv <file1> <file2>` : rename a file or directory from `<file1>` to `<file2>`
  - `mv <file> <directory>` : move a file or directory to a directory

- `mv <directory> <directory>` : move a directory to another directory
- `cat <file>` : print the content of a file

## Advanced commands

- `less <file>` : view a file with a read-only mode
- `diff <file1> <file2>` : compare two files
  - useful options:
    - `-y` : output in two columns
    - `-w` : ignore all white spaces, important when comparing two files with different indentation
- `nano` , `gedit` , `vim` , `emacs` : text editors

## I/O Redirection

In Linux, we can redirect the input and output of a command. The general syntax is:

```
command/executable < input > output
# e.g.
cat < input.txt > output.txt
./my_program < input.in > output.out
```

Here the `input` is used as `stdin` of the command/executable and the `output` is used as `stdout` of the command/executable. Note:

- The output redirection will overwrite the file if it exists
- The input redirection will fail if the file does not exist
- The output redirection will create the file if it does not exist
- To append to a file, use `>>` instead of `>`
  - e.g. `./my_executable >> output.out` will append the output to `output.out`

## Lecture 3: Develop and Compile Programs

### Compilation Process

- Preprocessing: The codes with `#` are processed. The `#include` directive is replaced by the content of the header file.

- Compilation: Compile the `.c/.cpp` files into `.o` files, or object files.
- Linking: Link the object files into an executable file.

## Compilation Commands & Makefile

Suppose you have a program with `my_program1.cpp` and `my_program2.cpp` and you want to compile them into an executable file `my_executable`, you can use the following commands to compile them:

```
g++ -o my_executable my_program1.cpp my_program2.cpp
```

These commands are actually integrated by the following steps:

1. `g++ -c my_program1.cpp`
2. `g++ -c my_program2.cpp`
3. `g++ -o my_executable my_program1.o my_program2.o`

Note that you don't need to manually do the preprocessing step by commands.

Also, you can use a `Makefile` to automate the compilation process. A `Makefile` is a file that contains a set of rules used to build a target file from source files. The general syntax of a rule is:

```
target: dependencies  
      command
```

The sample Makefile for the above example is:

```
all: my_executable  
  
my_executable: my_program.o my_class.o  
    g++ -o my_executable my_program.o my_class.o  
  
my_program.o: my_program.cpp  
    g++ -c my_program.cpp  
  
my_class.o: my_class.cpp  
    g++ -c my_class.cpp
```

```
clean:
    rm -f my_executable *.o # Here * is a wildcard
```

To use the Makefile to compile the program, type `make` in the terminal. To clean the compiled files, type `make clean`. The general rule is `make <target>`.

Pay attention to these two compile flags:

- `-c`: Used to compile the source files into object files
- `-o`: Used to specify the output file's name

## Header File and Header Guard

A header file is a file that contains declarations of functions, classes, and variables. The following is an example of a header file:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

void my_function();

#endif
```

- The functions declared in the header file should be defined in the corresponding `.cpp` file.

```
#include "my_header.h"

void my_function() {
    // function body
}
```

- The `#ifndef`, `#define`, and `#endif` are called header guards. They prevent the header file from being included multiple times in the same file.
- To use the header file in a `.cpp` file, include the header file at the beginning of the file.

```
#include "my_header.h"

int main() {
    my_function();
    return 0;
}
```

## Lecture 4: Review of C++ Basics

---

### Basic Operators

Just pay attentions to these two operators:

- `i++` vs. `++i`: The former returns the value of `i` before incrementing, while the latter returns the value of `i` after incrementing.

### lvalue and rvalue

- **lvalue**: an expression that may appear on the left-hand side or right-hand side of an assignment.
- **rvalue**: an expression that may **only** appear on the right-hand side of an assignment.

Examples:

```
int a = 1; // a is an lvalue, 1 is an rvalue
int b = a; // b is an lvalue, a is an lvalue
const int c = a + b; // c is an rvalue, a + b is an rvalue
int *p = &a; // p is an lvalue, &a is an rvalue
```

### Function Declaration and Definition

- **Declaration:**

```
// return_type function_name(parameter_list);
int add(int a, int b);
void print(string s);
```

Note that this should come before the function is called.

- **Definition:**

```
// return_type function_name(parameter_list) {  
//     function_body;  
// }  
  
int add(int a, int b) {  
    return a + b;  
}  
  
void print(string s) {  
    cout << s << endl;  
}
```

Note that this can come before or after the function is called.

## Pointers, References and Arrays

- **Pointers:** a variable that stores the address of another variable. Changing the value of a pointer will change the value of the variable it points to.

```
int a = 114514;  
int *p = &a; // p is a pointer to a, &a is the address of a  
cout << *p << endl; // *p is the value of a  
*p = 1919810; // a is now 1919810
```

- Key points:
  - `*` is the dereference operator, which returns the value of the variable that the pointer points to.
  - `&` is the address-of operator, which returns the address of a variable.
  - The address of a variable is rvalue. Like in the example above, you cannot change `&a`.
- **References:** an **alias** of a variable. Changing the value of a reference will change the value of the variable it refers to.

```
int a = 114514;  
int &r = a; // & is the reference operator, r is a reference to a
```

```
r = 1919810; // a is now 1919810
```

- Key points:
  - A reference must be initialized when it is declared.
  - After initialization, a reference cannot be changed to refer to another variable.

Make sure you understand the two examples below:

```
int x = 0;
int &r = x;
int y = 1;
r = y;
r = 2; // What is the value of x, y, and r?
```

```
int x = 0;
int *p = &x;
int y = 1;
p = &y;
*p = 2; // What is the value of x, y, and *p?
```

**Arrays:** a collection of variables of the same type. The size of an array must be known at compile time. You can access the elements of an array using `[]` operator or by pointers.

```
int a[5] = {1, 2, 3, 4, 5}; // a is an array of 5 integers
cout << a == &a[0] << endl; // true
cout << a[0] << endl; // 1
```

Arrays work well with pointers. It is actually a pointer to the first element of the array, and the next pointer is the address of the next element, and so on.

## Function Call Mechanism

When a function is called, the parameters are passed to the function. There are two ways to pass parameters:

- **Pass by value:** the value of the parameter is copied to the function. Modifying the parameter inside the function will not affect the original variable.



- **Pass by reference:** the address of the parameter is passed to the function. Modifying the parameter inside the function will affect the original variable.

A brief example:

```
void f(int x, int &y, int *z) {  
    x = 1; // passed by value  
    y = 1; // passed by reference  
    *z = 1; // passed by reference  
}
```

In C++, arrays are passed by reference.

```
void f(int a[]) {  
    a[0] = 1; // passed by reference, the value of a[0] is changed  
}
```

Advantage of pass by reference with **pointers**:

- You can directly change the value of the variable that the pointer points to. This **avoid copying instances** of large objects sometimes.

Advantage of pass by reference with **references**:

- More readable and intuitive than pointers.
- It keeps the advantage of avoid unnecessary copying.

## Structs

Structs are **user-defined data types** that can contain multiple variables of different types. The members of a struct can be common data types or other structs.

```
struct Student {  
    string name;  
    int id;  
    double gpa;  
}; // don't forget the semicolon here
```

To declare a struct variable and access its members:

```
Student s1; // s1 is a struct variable
s1.name = "Alice";
s1.id = 114514;
s1.gpa = 4.0;
Student s2 = {"Bob", 1919810, 3.9}; // another way to initialize
Student *s3 = &s2; // s3 is a pointer to s2
s3->name = "Jack"; // access the members of struct pointer using ->
```

## Lecture 5: Const Qualifier

---

### Const Data Variables

Basic syntax:

```
const data_type variable_name = value;
```

Properties:

- The value of a const variable cannot be changed after initialization.

```
const int MAX_SIZE = 100;
MAX_SIZE = 200; // Error
```

- A const variable must be initialized when it is declared.

```
const int MAX_SIZE; // Error
```

### Const References

Basic syntax:

```
const data_type &reference_name = variable_name;
const int &r = a;
```

Advantages:

- **Cannot be modified.**

Recall that the parameter can be passed to functions by reference to avoid copying large

objects. However, if you don't want the function to modify the parameter, you can pass it by const reference.

```
Student s1 = {"Alice", 12345, 1.0};  
// Recall the struct Student  
void f(const Student &s) {  
    s.name = "Alice";  
    s.gpa = 4.0;  
    cout << s.gpa << endl; // 4.0  
}
```

- **Can be initialized by rvalues.**

Recall that a reference must be initialized when it is declared. However, a const reference can be initialized by rvalues.

```
const int &cref = 1; // OK  
int &ref = 1; // Error
```

For function parameters, it is recommended to pass by const reference if the parameter is not modified inside the function. The type compatibility is as follows:

- `const type &` to `type &` is not allowed.
- `type &` to `const type &` is allowed.
- `const type *` to `type *` is not allowed.
- `type *` to `const type *` is allowed.

In one word, **only coercion from non-const to const is allowed.**

A small exercise:

```
void const_reference_test(const int &r) {}  
void reference_test(int &r) {}  
void pointer_test(int *p) {}  
int main() {  
    int a = 0;  
    const int b = 0;  
    int *p = &a;  
    const int *cp = &a;
```

```

// Which of the following function calls are valid?
const_reference_test(a);
const_reference_test(b);
const_reference_test(*p);
const_reference_test(*cp);
reference_test(a);
reference_test(b);
reference_test(*p);
reference_test(*cp);
pointer_test(p);
pointer_test(cp);
}

```

## Const Pointers

The rule for `const`: `const` applies to the thing on its left. If there is nothing on its left, it applies to the thing on its right.

For const pointers, there are two cases:

- Pointer to `const`: Here the pointer can point to arbitrary variables, but the value of the variable cannot be modified.
- `const` pointer: Here the pointer can only point to one variable, but the value of the variable can be modified.

The two cases can be combined.

```

const int* a           // a pointer to a constant integer
int const * a          // a pointer to a constant integer
int* const a           // a constant pointer to an integer
const int* const a     // a constant pointer to a constant integer
int const * const a    // a constant pointer to a constant integer

```

These declarations are all valid.

## typedef

`typedef` is used to create an alias for a data type. It is often used to simplify the

declaration of complex data types, just like using reference variables.

```
typedef int* int_ptr;
// int_ptr is an alias for int*
typedef const int_ptr const_int_ptr;
// the defined alias can be used in other typedef
typedef const int* const_int_ptr;
// All in one line
```

## Lecture 6: Procedure Abstraction

---

### Abstraction

Abstraction is a process of emphasizing the separation of "what" and "how". It helps programmers to use a function without knowing how it is implemented.

It only provides the details that are relevant to the user, and hide the unnecessary details.

```
// In the header file, the function is declared
int add(int a, int b);

// In the cpp file, the function is defined/implemented
int add(int a, int b) {
    return a + b;
}
```

### Properties of a Good Abstraction

- **Local:** The implementation of an abstraction is independent of any other abstraction implementation.

```
// Here the user doesn't need to know how multiply is implemented
int square(int a) {
    return multiply(a, a);
}
```

- **Substitutable:** The implementation of an abstraction can be replaced by another implementation as long as the interface is the same and the implementation is correct.

```
// Here the implementation of multiply can be replaced by another implementation
// as long as the abstraction is the same and the implementation is correct
int multiply(int a, int b) {
    return a * b;
    // return b * a; // This is also correct
}
```

## Type Signature

The type signature of a function is the function's name, the number of parameters, and the type of each parameter. It is used to declare the function.

```
int add(int a, int b);
```

## Specification Comments

Besides the type signature, a function should also have specification comments. There're usually three types of comments:

- **REQUIRES:** Preconditions that must hold, if any.
- **MODIFIES:** Variables that are modified, if any.
- **EFFECTS:** What the procedure does given legal inputs.

```
int positiveAdd(int a, int b);
// REQUIRES: a > 0, b > 0
// MODIFIES: None
// EFFECTS: Returns the sum of a and b
```

Note that functions with no **REQUIRES** clause is complete, while functions with them are partial.

## Reference

---

[1] Yancheng, Wu. VE280 Mid RC Part 1. 2023.

[2] Zhanxun, Liu. VE280 Mid RC Part 2. 2023.