VE280 2021FA RC2

created by Sun Jiajun 2020.9.24

(FYI: Just reading this PDF may be exhausting for you because most of the demonstration and explanation are along with the RC demo. If you are reviewing, please still refer to the professor's slides.)

L3: Developing Programs

Makefile (very important)

```
Target:Dependency
<Tab> Command
```

- Dependency: A list of files that the target depends on.

Here is an rather complete but complex example:

```
edit: main.o kbd.o command.o display.o /
      insert.o search.o files.o utils.o
    gcc -o edit main.o kbd.o command.o display.o /
              insert.o search.o files.o utils.o

main.o: main.c defs.h
    gcc -c main.c
kbd.o: kbd.c defs.h command.h
    gcc -c kbd.c
command.o: command.c defs.h command.h
    gcc -c command.c
display.o: display.c defs.h buffer.h
    gcc -c display.c
insert.o: insert.c defs.h buffer.h
    gcc -c insert.c
search.o: search.c defs.h buffer.h
    gcc -c search.c
files.o: files.c defs.h buffer.h command.h
    gcc -c files.c
utils.o: utils.c defs.h
    gcc -c utils.c
clean:
    rm edit main.o kbd.o command.o display.o /
            insert.o search.o files.o utils.o
```

How makefile works? (Just a basic review)

1. Make will look for a file named **"Makefile" or "Makefile"** in the current directory.

2. If found, it looks for the first target in the file. In the above example, it finds the **"edit" file** and treats it as the **final target file**.
3. If the Edit file does not exist, **or if the later.o file on which Edit depends is newer than the edit file**, it generates the edit file by executing the commands defined below.
4. If the.o file on which Edit depends also exists, Make looks for.o **dependencies** in the current file and generates.o files based on that rule.
5. Of course, your C and H files exist, so Make generates.o files and then uses the.o files. The final task of Make is to execute the edit file.

- All Target:
  - It is the default target.
  - Its dependency is program name.
  - It has no command.

Since the treats it as the final target fileTherefore, if you put the all at the first line, and "all:" has only the dependency program name.

- Clean Target:
  - “**make clean**”
  - It has no dependency!
  - what is -f?

Important: **Make** compares the date of the **dependency** with the **date** of the modification of the Targets file. If the Date of the dependency file is newer than that of the Targets file, or if target does not exist, **Make** executes the command defined later.

To consider: Do we need to include header files into the dependency files? (Please follow the demo)

Something useful but not covered: **How to use the variables in Makefile?**

To make Makefiles easy to maintain, we can use variables in Makefiles. The variable in a Makefile is simply a string.

You can name it objects, OBJECTS, objs, OBJS, obj, etc.

```
objects= main.o kbd.o command.o display.o /
              insert.o search.o files.o utils.o
```

```
objects = main.o kbd.o command.o display.o /
          insert.o search.o files.o utils.o

edit: $(objects)
    gcc -o edit $(objects)
main.o: main.c defs.h
    gcc -c main.c
```

```
kbd.o: kbd.c defs.h command.h
    gcc -c kbd.c
command.o: command.c defs.h command.h
    gcc -c command.c
display.o: display.c defs.h buffer.h
    gcc -c display.c
insert.o: insert.c defs.h buffer.h
    gcc -c insert.c
search.o: search.c defs.h buffer.h
    gcc -c search.c
files.o: files.c defs.h buffer.h command.h
    gcc -c files.c
utils.o: utils.c defs.h
    gcc -c utils.c
clean:
    rm edit $(objects)
```

Continue on

GNU Make is powerful enough to **automatically derive** the commands after files and file dependencies, so there is **no need to write similar commands at the end of every [.o] file**, because our make will automatically recognize and derive the commands itself.

Whenever make sees a [.o] file, it **automatically adds the [.c] file to the dependency**. If make finds an imitate. o file, then imitate. c will be the imitate. o dependency.

```
objects = main.o kbd.o command.o display.o /
              insert.o search.o files.o utils.o

edit: $(objects)
    cc -o edit $(objects)

main.o: defs.h
kbd.o: defs.h command.h
command.o: defs.h command.h
display.o: defs.h buffer.h
insert.o: defs.h buffer.h
search.o: defs.h buffer.h
files.o: defs.h buffer.h command.h
utils.o: defs.h

.PHONY: clean
clean :
    rm edit $(objects)
```

Can we also omit some of the [.h] files? Of Course.

```
objects = main.o kbd.o command.o display.o /
      insert.o search.o files.o utils.o
```

```
edit: $(objects)
    cc -o edit $(objects)

$(objects): defs.h
kbd.o command.o files.o: command.h
display.o insert.o search.o files.o: buffer.h

.PHONY: clean
clean:
    rm edit $(objects)
```

For more information: https://blog.csdn.net/haoel/article/details/2886

Header Guard (revisit)

Including of a header file more than once may cause multiple definitions of the classes and functions defined in the header file.

With a header guard, we guarantee that the definition in the header is just seen once.

```
//add.h
#ifndef ADD_H // test whether ADD_H has not been defined before
#define ADD_H
int add(int a, int b);
#endif
```

Excercise:

(Adapted from https://www.freesion.com/article/7083439062/)

Step1: Create five files: **main.cpp, study.cpp, touch_fish.cpp, study.h, touch_fish.h.** Achieve the function in main that user's input of 0 or 1 decides whether to call **study()** or **touchfish()** function respectively written in study.cpp and touch_fish.cpp. Both fuctions only cout one line: **"study VE280 today"** or **"touched fish today! Study VE280 tomorrow!"** I think you can tell which is which 😃

Step2: Write the **Makefile by yourselves**. Please first finish the **complete version** (similar to complete but complex one above), because **the exam is based on this version**. After you finish, you can try the methods above to revise it.

L4 Review of C++ Basics

- Variables
    - Built-in data types, e.g., int, double, etc.
    - Input and output, e.g., cin, cout.
- Operators
    - Arithmetic: +, -, *, etc.

  - Comparison: <, >, ==, etc.
  - x++ versus ++x

- Notes:

  - **lvalue:** An expression which may appear as either the left-hand or right-hand side of an assignment
  - **rvalue:** An expression which may appear on the right- but not left-hand side of an assignment

For C++ basics, most of them are covered by the slides and you already know then by heart, but feel free to test the following codes by yourself or come to the RC to watch the demo.

- pointer:
  - They provide a convenient mechanism to work with arrays.
  - They allow us to create structures (unlike arrays) whose size is not known in advance

```
int a[2][2] = {{1, 2}, {3, 4}};
    int b[2][2] = {1, 2, 3, 4};
    int *ptr = b[0];
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0 ; j < 2; j++)
        {
            cout << i << "," << j << ":" << a[i][j] << ","
            << b[i][j] << "," << *(ptr + i * 2 + j) << endl;
        }
    }
```

- Reference:

1. Reference must be **initialized** using a variable of the same type.

2. There is no way to rebind a reference to a different object

```
    int a = 10;
    //give the variable a an other name "b"
    int& b = a;
    cout << "a:" << a << endl;
    cout << "b:" << b << endl;
    cout << "------------" << endl;
    //doing operation on b is the same as doing operation on a
    b = 100;
    cout << "a:" << a << endl;
    cout << "b:" << b << endl;
    cout << "------------" << endl;
    //an variable can have many other names
    int& c = a;
    c = 200;
    cout << "a:" << a << endl;
    cout << "b:" << b << endl;
    cout << "c:" << c << endl;
    cout << "------------" << endl;
    //the address for a,b,c are all the same
    cout << "a:" << &a << endl;
    cout << "b:" << &b << endl;
    cout << "c:" << &c << endl;
```

- struct:

```
struct Grades{
char name[9];
int midterm;
int final;
};

#include <iostream>
using namespace std;

int main(){
   struct Grades alice= {"Alice", 60, 85};
    alice.midterm=65;
    struct Grades *gPtr = &alice;
    gPtr->final = 90;
    cout << alice.name << "'s midterm is " <<
    alice.midterm << " and final is " << alice.final << endl;
    cout << gPtr->name << "'s midterm is " <<
    gPtr->midterm << " and final is " << gPtr->final << endl;
}
```