Separate Compilation and Makefiles

Separate Compilation

Slides created by David Mann, North Idaho College

from
Problem Solving with C++:
The Art of Programming, 4th edition
By Walter Savitch

Separate Compilation

- C++ allows you to divide a program into parts
  - Each part can be stored in a separate file
  - Each part can be compiled separately
  - A class definition can be stored separately from a program.
    - This allows you to use the class in multiple programs

Copyright © 2003 Pearson Education, Inc.

ADT Review

- An ADT is a class defined to separate the interface and the implementation
  - All member variables are private
  - The class definition along with the function and operator declarations are grouped together as the interface of the ADT
  - Group the implementation of the operations together and make them unavailable to the programmer using the ADT

Copyright © 2003 Pearson Education, Inc.

The ADT Interface

- The **interface** of the ADT includes
  - The class definition
  - The declarations of the basic operations which can be one of the following
    - Public member functions
    - Friend functions
    - Ordinary functions
    - Overloaded operators
  - The function comments

Copyright © 2003 Pearson Education, Inc.

The ADT Implementation

- The **implementation** of the ADT includes
  - The function definitions
    - The public member functions
    - The private member functions
    - Non-member functions
    - Private helper functions
  - Overloaded operator definitions
  - Member variables
  - Other items required by the definitions

Copyright © 2003 Pearson Education, Inc.

Separate Files

- In C++ the ADT interface and implementation can be stored in separate files
  - The **interface file** stores the ADT interface
  - The **implementation file** stores the ADT implementation

Copyright © 2003 Pearson Education, Inc.

A Minor Compromise

- The public part of the class definition is part of the ADT interface
- The private part of the class definition is part of the ADT implementation
  - This would hide it from those using the ADT
- C++ does not allow splitting the public and private parts of the class definition across files
  - The entire class definition is usually in the interface file

Copyright © 2003 Pearson Education, Inc.

Case Study: DigitalTime

- The interface file of the DigitalTime ADT class contains the class definition
  - The values of the class are:
    - Time of day, such as 9:30, in 24 hour notation
  - The public members are part of the interface
  - The private members are part of the implementation
  - The comments in the file should provide all the details needed to **use** the ADT

Copyright © 2003 Pearson Education, Inc.

Naming The Interface File

- The DigitalTime ADT interface is stored in a file named **dtime.h**
  - The .h suffix means this is a header file
  - Interface files are always header files
- A program using dtime.h must include it using an include directive

```
#include "dtime.h"
```

Display 9.1

Copyright © 2003 Pearson Education, Inc.

Display 9.1

Interface File for `DigitalTime`

```
//Header file dtime.h: This is the INTERFACE for the class DigitalTime.
//Values of this type are times of day. The values are input and output in
//24-hour notation, as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.
#include <iostream>          // For the definition of the types
using namespace std;         // istream and ostream, which
                             // are used as parameter types
class DigitalTime
{
public:
    friend bool operator ==(const DigitalTime& time1, const DigitalTime& time2);
    //Returns true if time1 and time2 represent the same time;
    //otherwise, returns false.

    DigitalTime(int the_hour, int the_minute);
    //Precondition: 0 <= the_hour <= 23 and 0 <= the_minute <= 59.
    //Initializes the time value to the_hour and the_minute.

    DigitalTime( );
    //Initializes the time value to 0:00 (which is midnight).

    void advance(int minutes_added);
    //Precondition: The object has a time value.
    //Postcondition: The time has been changed to minutes_added minutes later.

    void advance(int hours_added, int minutes_added);
    //Precondition: The object has a time value.
    //Postcondition: The time value has been advanced
    //hours_added hours plus minutes_added minutes.

    friend istream& operator >>(istream& ins, DigitalTime& the_object);
    //Overloads the >> operator for input values of type DigitalTime.
    //Precondition: If ins is a file input stream, then ins has already been
    //connected to a file.

    friend ostream& operator <<(ostream& outs, const DigitalTime& the_object);
    //Overloads the << operator for output values of type DigitalTime.
    //Precondition: If outs is a file output stream, then outs has already been
    //connected to a file.
private:
    int hour;        // This is part of the implementation.
    int minute;      // It is not part of the interface.
                     // The word private indicates that
                     // this is not part of the public interface.
};
```

Copyright © 2003 Pearson Education, Inc.

#include " " or < > ?

- To include a predefined header file use < and >

  #include <iostream>

  - < and > tells the compiler to look where the system stores predefined header files
- To include a header file you wrote, use " and "

  #include "dtime.h"

  - " and " usually cause the compiler to look in the current directory for the header file

Copyright © 2003 Pearson Education, Inc.

The Implementation File

- Contains the definitions of the ADT functions
- Usually has the same name as the header file but a different suffix
  - Since our header file is named dtime.h, the implementation file is named **dtime.cpp**
  - Suffix depends on your system
    (some use .cxx or .CPP)

Copyright © 2003 Pearson Education, Inc.

#include "dtime.h"

- The implementation file requires an include directive to include the interface file:

#include "dtime.h"

Display 9.2 (1)
Display 9.2 (2)
Display 9.2 (3)
Display 9.2 (4)

Copyright © 2003 Pearson Education, Inc.

Display 9.2 (1/4)

Implementation File for `DigitalTime` (*part 1 of 4*)

```
//Implementation file dtime.cpp (Your system may require some
//suffix other than .cpp): This is the IMPLEMENTATION of the ADT DigitalTime.
//The interface for the class DigitalTime is in the header file dtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "dtime.h"
using namespace std;

//These FUNCTION DECLARATIONS are for use in the definition of
//the overloaded input operator >>:

void read_hour(istream& ins, int& the_hour);
//Precondition: Next input in the stream ins is a time in 24-hour notation,
//like 9:45 or 14:45.
//Postcondition: the_hour has been set to the hour part of the time.
//The colon has been discarded and the next input to be read is the minute.

void read_minute(istream& ins, int& the_minute);
//Reads the minute from the stream ins after read_hour has read the hour.

int digit_to_int(char c);
//Precondition: c is one of the digits '0' through '9'.
//Returns the integer for the digit; for example, digit_to_int('3') returns 3.

bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
{
    return (time1.hour == time2.hour && time1.minute == time2.minute);
}

//Uses iostream and cstdlib:
DigitalTime::DigitalTime(int the_hour, int the_minute)
{
    if (the_hour < 0 || the_hour > 23 || the_minute < 0 || the_minute > 59)
    {
        cout << "Illegal argument to DigitalTime constructor.";
        exit(1);
    }
```

Copyright © 2003 Pearson Education, Inc.

Display 9.2 (2/4)

Implementation File for DigitalTime (*part 2 of 4*)

```
    else
    {
        hour = the_hour;
        minute = the_minute;
    }
}

DigitalTime::DigitalTime( ) : hour(0), minute(0)
{
    //Body intentionally empty.
}

void DigitalTime::advance(int minutes_added)
{
    int gross_minutes = minute + minutes_added;
    minute = gross_minutes%60;

    int hour_adjustment = gross_minutes/60;
    hour = (hour + hour_adjustment)%24;
}

void DigitalTime::advance(int hours_added, int minutes_added)
{
    hour = (hour + hours_added)%24;
    advance(minutes_added);
}

//Uses iostream:
ostream& operator <<(ostream& outs, const DigitalTime& the_object)
{
    outs << the_object.hour << ':';
    if (the_object.minute < 10)
        outs << '0';
    outs << the_object.minute;
    return outs;
}
```

Copyright © 2003 Pearson Education, Inc.

Display 9.2 (3/4)

Implementation File for DigitalTime (*part 3 of 4*)

```
//Uses iostream:
istream& operator >>(istream& ins, DigitalTime& the_object)
{
    read_hour(ins, the_object.hour);
    read_minute(ins, the_object.minute);
    return ins;
}

int digit_to_int(char c)
{
    return ( int(c) - int('0') );
}

//Uses iostream, cctype, and cstdlib:
void read_minute(istream& ins, int& the_minute)
{
    char c1, c2;
    ins >> c1 >> c2;

    if (!(isdigit(c1) && isdigit(c2)))
    {
        cout << "Error illegal input to read_minute\n";
        exit(1);
    }

    the_minute = digit_to_int(c1)*10 + digit_to_int(c2);

    if (the_minute < 0 || the_minute > 59)
    {
        cout << "Error illegal input to read_minute\n";
        exit(1);
    }
}
```

Copyright © 2003 Pearson Education, Inc.

Display 9.2 (4/4)

Implementation File for `DigitalTime` (*part 4 of 4*)

```
//Uses iostream, cctype, and cstdlib:
void read_hour(istream& ins, int& the_hour)
{
    char c1, c2;
    ins >> c1 >> c2;
    if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )
    {
        cout << "Error illegal input to read_hour\n";
        exit(1);
    }

    if (isdigit(c1) && c2 == ':')
    {
        the_hour = digit_to_int(c1);
    }
    else //(isdigit(c1) && isdigit(c2))
    {
        the_hour = digit_to_int(c1)*10 + digit_to_int(c2);
        ins >> c2;//discard ':'
        if (c2 != ':')
        {
            cout << "Error illegal input to read_hour\n";
            exit(1);
        }
    }

    if ( the_hour < 0 || the_hour > 23 )
    {
        cout << "Error illegal input to read_hour\n";
        exit(1);
    }
}
```

Copyright © 2003 Pearson Education, Inc.

The Application File

- The Application file is the file that contains the program that uses the ADT
  - It is also called a driver file
  - Must use an include directive to include the interface file:

```
#include "dtime.h"
```

Display 9.3

Copyright © 2003 Pearson Education, Inc.

Display 9.3

Application File Using `DigitalTime`

```
//Application file timedemo.cpp (your system may require some suffix
//other than .cpp): This program demonstrates use of the class DigitalTime.
#include <iostream>
#include "dtime.h"
using namespace std;

int main( )
{
    DigitalTime clock, old_clock;

    cout << "Enter the time in 24-hour notation: ";
    cin >> clock;

    old_clock = clock;
    clock.advance(15);
    if (clock == old_clock)
        cout << "Something is wrong.";
    cout << "You entered " << old_clock << endl;
    cout << "15 minutes later the time will be "
         << clock << endl;

    clock.advance(2, 15);
    cout << "2 hours and 15 minutes after that\n"
         << "the time will be "
         << clock << endl;

    return 0;
}
```

Sample Dialogue

```
Enter the time in 24-hour notation: 11:15
You entered 11:15
15 minutes later the time will be 11:30
2 hours and 15 minutes after that
the time will be 13:45
```

Copyright © 2003 Pearson Education, Inc.

Running The Program

- Basic steps required to run a program:
(Details vary from system to system!)
  - Compile the implementation file
  - Compile the application file
  - Link the files to create an executable program using a utility called a **linker**
    - Linking is often done automatically

Copyright © 2003 Pearson Education, Inc.

Compile dtime.h ?

- The interface file is not compiled separately
  - The preprocessor replaces any occurrence of #include "dtime.h" with the text of dtime.h before compiling
  - Both the implementation file and the application file contain #include "dtime.h"
    - The text of dtime.h is seen by the compiler in each of these files
    - There is no need to compile dtime.h separately

Copyright © 2003 Pearson Education, Inc.

Why Three Files?

- Using separate files permits
  - The ADT to be used in other programs without rewriting the definition of the class for each
  - Implementation file to be compiled once even if multiple programs use the ADT
  - Changing the implementation file does not require changing the program using the ADT

Copyright © 2003 Pearson Education, Inc.

Reusable Components

- An ADT coded in separate files can be used over and over
- The **reusability** of such an ADT class
  - Saves effort since it does not need to be
    - Redesigned
    - Recoded
    - Retested
  - Is likely to result in more reliable components

Copyright © 2003 Pearson Education, Inc.

Multiple Classes

- A program may use several classes
  - Each could be stored in its own interface and implementation files
    - Some files can "include" other files, that include still others
  - It is possible that the same interface file could be included in multiple files
  - C++ does not allow multiple declarations of a class
  - The **#ifndef** directive can be used to prevent multiple declarations of a class

Copyright © 2003 Pearson Education, Inc.

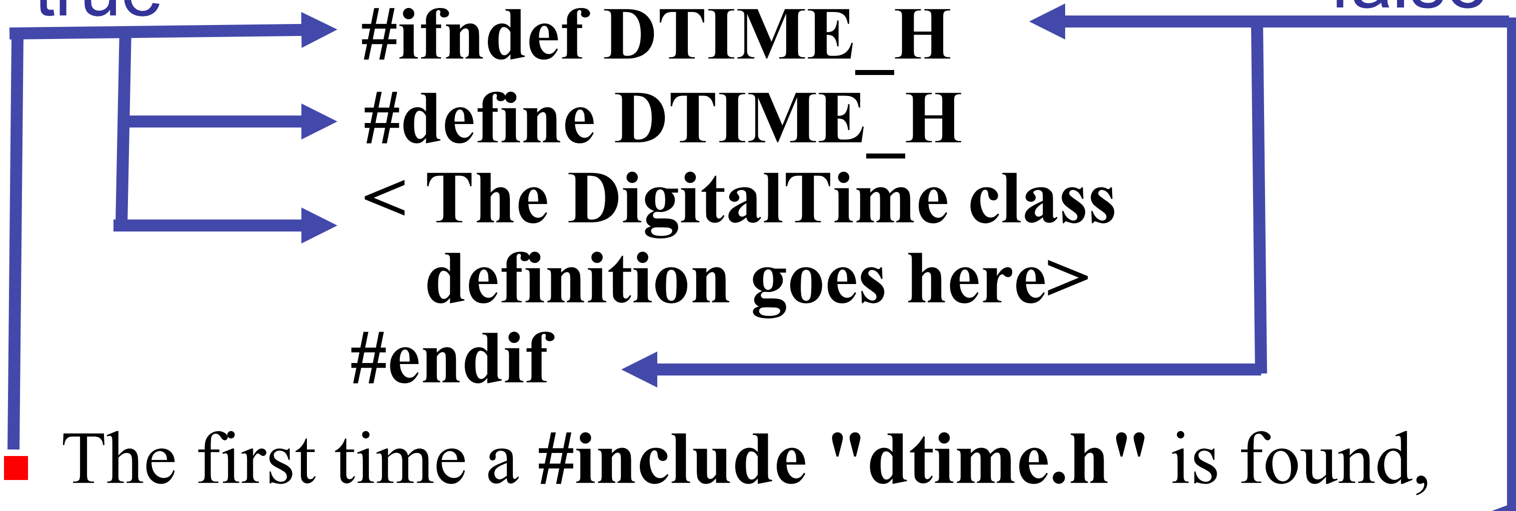
Introduction to #ifndef

- To prevent multiple declarations of a class, we can use these directives:
  - **#define DTIME_H**
    adds DTIME_H to a list indicating DTIME_H has been seen
  - **#ifndef DTIME_H**
    checks to see if DTIME_H has been defined
  - **#endif**
    If DTIME_H has been defined, skip to #endif

Copyright © 2003 Pearson Education, Inc.

Using #ifndef

- Consider this code in the interface file:



  - The first time a **#include "dtime.h"** is found, DTIME_H and the class are defined
  - The next time a **#include "dtime.h"** is found, all lines between #ifndef and #endif are skipped

Copyright © 2003 Pearson Education, Inc.

Why DTIME_H?

- DTIME_H is the normal convention for creating an identifier to use with ifndef
  - It is the file name in all caps
  - Use ' _ ' instead of ' . '
- You may use any other identifier, but will make your code more difficult to read

Display 9.4

Copyright © 2003 Pearson Education, Inc.

Display 9.4

Avoiding Multiple Definitions of a Class

```
//Header file dtime.h: This is the INTERFACE for the class DigitalTime.
//Values of this type are times of day. The values are input and output in
//24-hour notation, as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.

#ifndef DTIME_H
#define DTIME_H

#include <iostream>
using namespace std;

class DigitalTime
{

<The definition of the class DigitalTime is the same as in Display 9.1.>

};

#endif //DTIME_H
```

Copyright © 2003 Pearson Education, Inc.

Defining Libraries

- You can create your own libraries of functions
  - You do not have to define a class to use separate files
  - If you have a collection of functions…
    - Declare them in a header file with their comments
    - Define them in an implementation file
    - Use the library files just as you use your class interface and implementation files

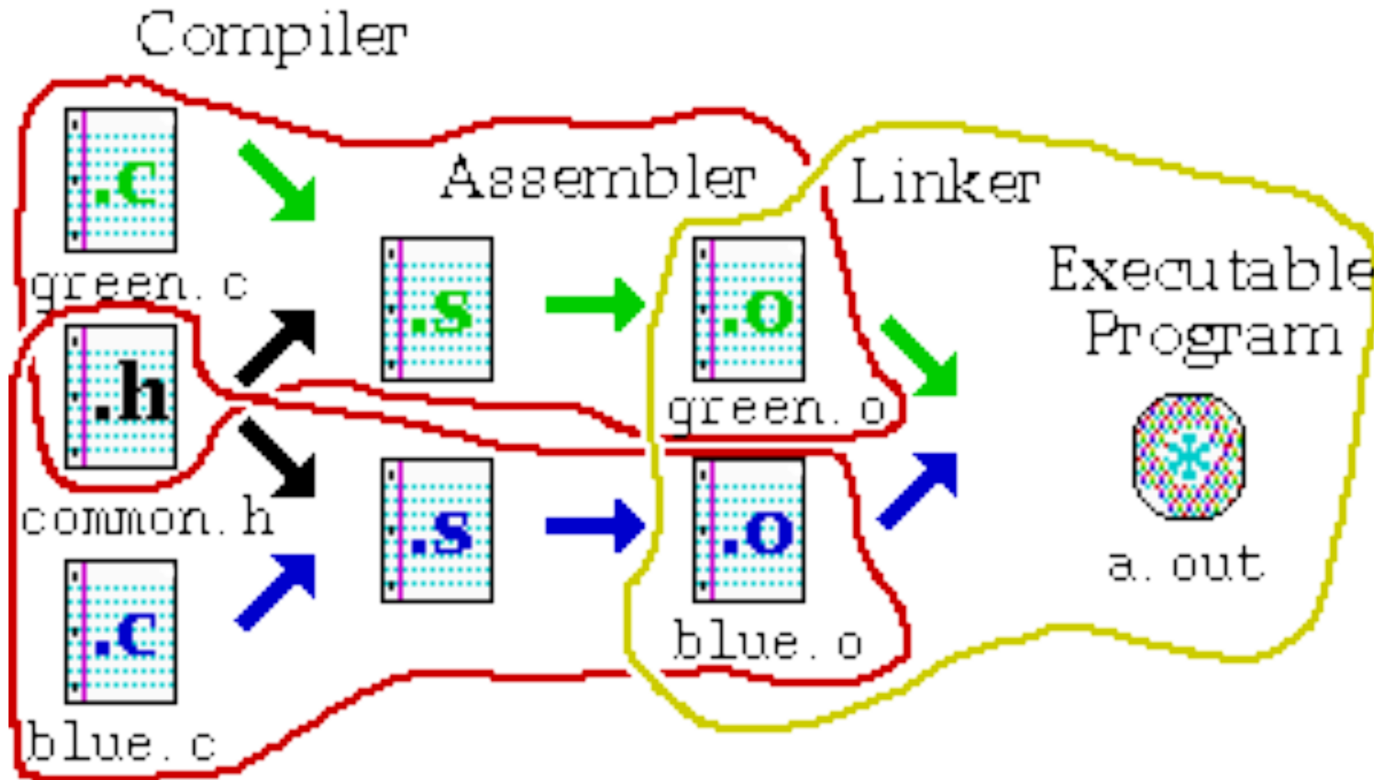
Copyright © 2003 Pearson Education, Inc.

Makefiles

Based on slides by George Bebis (U Nevada, Reno)

Makefiles

- Provide a way for separate compilation.
- Describe the dependencies among the project files.
- The `make` utility.



Using makefiles

Naming:

- *makefile* or *Makefile* are standard
- other name can be also used

Running `make`

`make`

`make -f filename` – if the name of your file is not "makefile" or "Makefile"

`make target_name` – if you want to make a target that is not the first one

Sample makefile

- Makefiles main element is called a *rule*:

```
target : dependencies
TAB   commands                    #shell commands
```

Example:

```
my_prog : eval.o main.o
  g++ -o my_prog eval.o main.o

eval.o : eval.c eval.h
  g++ -c eval.c
main.o : main.c eval.h
  g++ -c main.c
```

____

```
# -o to specify executable file name
# -c to compile only (no linking)
```

Variables

The old way (no variables)

```
my_prog : eval.o main.o
        g++ -o my_prog eval.o main.o
eval.o : eval.c eval.h
        g++ -c -g eval.c
main.o : main.c eval.h
        g++ -c -g main.c
```

A new way (using variables)

```
C = g++
OBJS = eval.o main.o
HDRS = eval.h

my_prog : eval.o main.o
        $(C) -o my_prog $(OBJS)
eval.o : eval.c
        $(C) -c -g eval.c
main.o : main.c
        $(C) -c -g main.c
$(OBJS) : $(HDRS)
```

<u>Defining variables on the command line:</u>

Take precedence over variables defined in the makefile.

```
make C=cc
```

Implicit rules

- Implicit rules are standard ways for making one type of file from another type.
- There are numerous rules for making an ***.o*** file – from a ***.c*** file, a ***.p*** file, etc. `make` applies the first rule it meets.
- If you have not defined a rule for a given object file, `make` will apply an implicit rule for it.

Example:

Our makefile

```
my_prog : eval.o main.o
   $(C) -o my_prog $(OBJS)
$(OBJS) : $(HEADERS)
```

The way `make` understands it

```
my_prog : eval.o main.o
        $(C) -o my_prog $(OBJS)
$(OBJS) : $(HEADERS)
eval.o : eval.c
        $(C) -c eval.c
main.o : main.c
        $(C) -c main.c
```

Automatic variables

Automatic variables are used to refer to specific part of rule components.

```
target : dependencies
TAB    commands              #shell commands
```

```
eval.o : eval.c eval.h
 g++ -c eval.c
```

`$@` - The name of the target of the rule (`eval.o`).

`$<` - The name of the first dependency (`eval.c`).

`$^` - The names of all the dependencies (`eval.c eval.h`).

`$?` - The names of all dependencies that are newer than the target

Defining implicit rules

```
%.o : %.c
   $(C) -c -g $<

C = g++
OBJS = eval.o main.o
HDRS = eval.h

my_prog : eval.o main.o
   $(C) -o my_prog $(OBJS)
$(OBJS) : $(HDRS)
```

Avoiding implicit rules - empty commands

`target: ;` #Implicit rules will not apply for this target.

`make` options

`make` options:

`-f` *filename* – when the makefile name is not standard

`-t` – (touch) mark the targets as up to date

`-q` – (question) are the targets up to date, exits with 0 if true

`-n` – print the commands to execute but do not execute them

/ `-t`, `-q`, and `-n`, cannot be used together /

`-s` – silent mode

`-k` – keep going – compile all the prerequisites even if not able to link them **!!**

Phony targets

Phony targets:

Targets that have no dependencies. Used only as names for commands that you want to execute.

```
clean :
 rm $(OBJS)
```

or

```
.PHONY : clean
clean:
        rm $(OBJS)
```

To invoke it: `make clean`

Typical phony targets:

`all` – make all the top level targets

```
.PHONY : all
all: my_prog1 my_prog2
```

`clean` – delete all files that are normally created by `make`

`print` – print listing of the source files that have changed

VPATH

- `VPATH` variable – defines directories to be searched if a file is not found in the current directory.

  `VPATH = `*`dir`*` : `*`dir`*` …`

  `/ VPATH = src:../headers /`

- `vpath` directive (lower case!) – more selective directory search:

  `vpath `*`pattern directory`*

  `/ vpath %.h headers /`

- `GPATH:`

  `GPATH` – if you want targets to be stored in the same directory as their dependencies.

Variable modifiers

```
C = g++
OBJS = eval.o main.o
SRCS = $(OBJS, .o=.c)     #!!!

my_prog : $(OBJS)
   $(C) -g -c $^

%.o : %.c
   $(C) -g -c S<

$(SRCS) : eval.h
```

Conditionals (directives)

Possible conditionals are:

`if` `ifeq` `ifneq` `ifdef` `ifndef`

All of them should be closed with `endif`.

Complex conditionals may use `elif` and `else`.

<u>Example:</u>

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)            #no tabs at the beginning
else
  libs=$(normal_libs)             #no tabs at the beginning
endif
```