Многофайловый проект. Автоматизация сборки проекта.

Недостатки однофайловых проектов

- Одновременная работа над программой нескольких программистов становится неэффективной.
- Ориентирование в тексте программы становится сложным.
- Даже при локальном изменении перекомпилируется весь проект.

Преимущества многофайловой организации проекта

- Позволяет распределить работу над проектом между несколькими программистами.
- Код программы более удобочитаем.
- Сокращает время повторной компиляции.
- Повторное использование кода.

Компиляция многофайлового проекта

```
// hello.c
#include <stdio.h>

void hello(void)
{
    printf("Hello!\n");
}
```

```
// main.c
int main(void)
{
    hello();

    return 0;
}
```

1. Напишите команду компиляции для каждого файла.
2. Возникнут ли ошибки во время компиляции?

Компиляция многофайлового проекта

```
// hello.c
#include <stdio.h>

void hello(void)
{
    printf("Hello!\n");
}
```

```
// main.c
int main(void)
{
    hello();

    return 0;
}
```

```
c99 -Wall -Werror -pedantic -c hello.c
c99 -Wall -Werror -pedantic -c main.c
```

Компиляция многофайлового проекта

```
main.c: В функции «main»:
main.c:3:5: ошибка: неявная декларация функции «hello»
[-Werror=implicit-function-declaration]
cc1.exe: все предупреждения считать ошибками
```

3. Как исправить ошибку?

Компиляция многофайлового проекта

```
// hello.c
#include <stdio.h>

void hello(void)
{
    printf("Hello!\n");
}
```

```
// main.c

void hello(void);

int main(void)
{
    hello();

    return 0;
}
```

Компиляция многофайлового проекта

```
c99 -o hello.exe hello.o

../libmingw32.a(main.o): In function `main':
../mingw/main.c:73: undefined reference to `WinMain@16'
collect2: выполнение ld завершилось с кодом возврата 1
==========================================================

c99 -o hello.exe main.o

main.o:main.c:(.text+0xc): undefined reference to `hello'
collect2: выполнение ld завершилось с кодом возврата 1
==========================================================

c99 -o hello.exe hello.o main.o
```

Заголовочные файлы

```
// hello.c
#include <stdio.h>

void hello(void)
{
    printf("Hello!\n");
}
```

```
// hello.h
void hello(void);
```

```
// main.c
#include “hello.h”

int main(void)
{
    hello();

    return 0;
}
```

Заголовочные файлы

```
// list.h
struct list_node
{
    void *data;
    struct list_node *next;
};

// ...
```

```
// hash.h
#include "list.h"

// ...

// hash.c
#include "list.h"
#include "hash.h"

// ...
```

```
c99 -Wall -Werror –pedantic -c hash.c

In file included from hash.h:1:0,
                 from hash.c:2:
list.h:1:8: ошибка: повторное определение «struct list_node»
list.h:1:8: замечание: originally defined here
```

Заголовочные файлы

```
// list.h

#ifndef __LIST_H__
#define __LIST_H__

struct list_node
{
    void *data;
    struct list_node *next;
};

// ...

#endif // __LIST_H__
```

```
// list.h

#pragma once

struct list_node
{
    void *data;
    struct list_node *next;
};

// ...
```

«Большой» проект

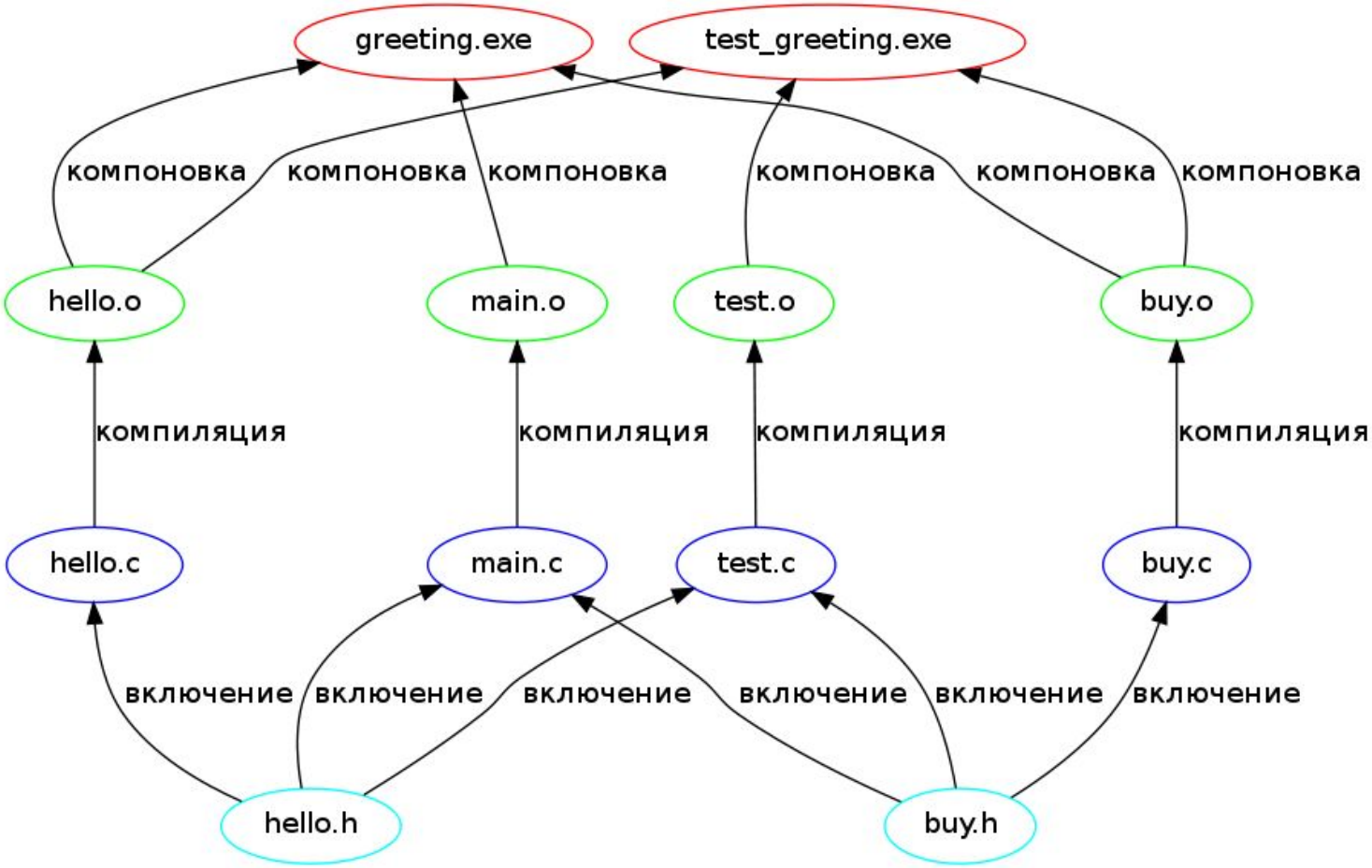
Компиляция

```
c99 -Wall -Werror -pedantic -c hello.c
c99 -Wall -Werror -pedantic -c buy.c
c99 -Wall -Werror -pedantic -c main.c
c99 -Wall -Werror -pedantic -c test.c
```

Компоновка

```
c99 -o greeting.exe hello.o buy.o main.o
c99 -o test_greeting.exe hello.o buy.o test.o
```

Граф зависимостей



Утилита make

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую.

- GNU Make (рассматривается далее)
- BSD Make
- Microsoft Make (nmake)

Утилита make: принципы работы

Необходимо создать так называемый сценарий сборки проекта (make-файл). Этот файл описывает

- отношения между файлами программы;
- содержит команды для обновления каждого файла.

Утилита make использует информацию из make-файла и время последнего изменения каждого файла для того, чтобы решить, какие файлы нужно обновить.

Утилита make предполагает, что по умолчанию сценарий сборки называется makefile или Makefile.

Сценарий сборки проекта

цель: зависимость_1 ... зависимость_n

[tab]команда_1

[tab]команда_2

...

[tab]команда_m

Простой сценарий сборки

```
greeting.exe : hello.o buy.o main.o
   gcc -o greeting.exe hello.o buy.o main.o

test_greeting.exe : hello.o buy.o test.o
   gcc -o test_greeting.exe hello.o buy.o test.o

hello.o : hello.c hello.h
   gcc -std=c99 -Wall -Werror -pedantic -c hello.c

buy.o : buy.c buy.h
   gcc -std=c99 -Wall -Werror -pedantic -c buy.c

main.o : main.c hello.h buy.h
   gcc -std=c99 -Wall -Werror -pedantic -c main.c

test.o : test.c hello.h buy.h
   gcc -std=c99 -Wall -Werror -pedantic -c test.c

clean :
   rm *.o *.exe
```

Использование переменных и комментариев

Строки, которые начинаются с символа #, являются комментариями.

Определить переменную в make-файле можно следующим образом:

```
VAR_NAME := value
```

Чтобы получить значение переменной, необходимо ее имя заключить в круглые скобки и перед ними поставить символ '$'.

```
$(VAR_NAME)
```

Использование переменных и комментариев

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o buy.o

greeting.exe : $(OBJS) main.o
   $(CC) -o greeting.exe $(OBJS) main.o

test_greeting.exe : $(OBJS) test.o
   $(CC) -o test_greeting.exe $(OBJS) test.o
```

Использование переменных и комментариев

```
hello.o : hello.c hello.h
   $(CC) $(CFLAGS) -c hello.c

buy.o : buy.c buy.h
   $(CC) $(CFLAGS) -c buy.c

main.o : main.c hello.h buy.h
   $(CC)  $(CFLAGS) -c main.c

test.o : test.c hello.h buy.h
   $(CC)  $(CFLAGS) -c test.c

clean :
   $(RM) *.o *.exe
```

Автоматические переменные

Автоматические переменные - это переменные со специальными именами, которые «автоматически» принимают определенные значения перед выполнением описанных в правиле команд.

- Переменная "$^" означает "список зависимостей".
- Переменная "$@" означает "имя цели".
- Переменная "$<" является просто первой зависимостью.

Было

```
greeting.exe : $(OBJS) main.o
   gcc -o greeting.exe $(OBJS) main.o
```

Стало

```
greeting.exe : $(OBJS) main.o
   gcc $^ -o $@
```

Автоматические переменные

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o buy.o

greeting.exe : $(OBJS) main.o
   $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
   $(CC) $^ -o $@
```

Автоматические переменные

```
hello.o : hello.c hello.h
   $(CC) $(CFLAGS) -c $<

buy.o : buy.c buy.h
   $(CC) $(CFLAGS) -c $<

main.o : main.c hello.h buy.h
   $(CC) $(CFLAGS) -c $<

test.o : test.c hello.h buy.h
   $(CC) $(CFLAGS) -c $<

clean :
   $(RM) *.o *.exe
```

Шаблонные правила

%.расш_файлов_целей : %.расш_файлов_зав

[tab]команда_1

[tab]команда_2

...

[tab]команда_m

Шаблонные правила

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o buy.o

greeting.exe : $(OBJS) main.o
   $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
   $(CC) $^ -o $@

%.o : %.c *.h
   $(CC) $(CFLAGS) -c $<

clean :
   $(RM) *.o *.exe
```

Сборка программы с разными параметрами компиляции

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o buy.o

ifeq ($(mode), debug)
    # Отладочная сборка: добавим генерацию отладочной информации
    CFLAGS += -g3
endif

ifeq ($(mode), release)
```

Сборка программы с разными параметрами компиляции

```
    # Финальная сборка: исключим отладочную информацию и утверждения (asserts)
    CFLAGS += -DNDEBUG -g0
endif

greeting.exe : $(OBJS) main.o
   $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
   $(CC) $^ -o $@

%.o : %.c *.h
   $(CC) $(CFLAGS) -c $<

clean :
   $(RM) *.o *.exe
```

Литература

1. Черновик стандарта C99
2. Б. Керниган, Д. Ритчи Язык программирования C
3. Артур Гриффитс, GCC: Настольная книга пользователей, программистов и системных администраторов.
4. Различные циклы уроков (tutorials) по make (например, http://habrahabr.ru/post/211751)