## Модульність

Зазвичай, в учбових курсах розглядаються лише прості та невеликі за обсягом програми, які містять декілька функцій та розміщуються в одному файлі. Однак в реальній програмістський роботі та в програмних доданках, що вирішують реальні практичні задачі, часто використовується великий обсяг тексту та сам програмний доданок складається з сотен а то й тисяч функцій. Досвід показує, що записувати всі функції програми в один файл незручно, оскільки зі збільшенням програми в такому файлі стає важко орієнтуватися, а також через те, що при цьому унеможливлюється спільна робота кількох програмістів. Тому грамотна технологія програмування полягає в тому, щоб розбивати велику програму на частини (так звані модулі), кожен з яких складається з порівняно невеликої кількості функцій.

В першому наближенні модуль являє собою окремий файл, в якому містяться визначення, описи структур та класів, тіла декількох функцій.

Переваги модульності, однак, зовсім не вичерпуються згаданою зручністю від розбиття великої кількості функцій на невеликі групи. Програміст може написати такий набір функцій, який може стати корисним для інших програмістів та коли ці функції зроблено достатньо гнучкими та універсальними, щоб їх можна було вставляти в чужі програми. як наслідок, модулі можуть бути самостійним різновидом товару на ринку програмного забезпечення.

Іншими словами, робота програміста може полягати не лише в тому, щоб виготовляти програми для кінцевого споживача, але й в тому, щоб писати модулі для інших програмістів, які будуть вставляти їх у свої програми.

Крім того, до складу одного програмного продукту цілком можуть входити модулі, написані різними мовами програмування. Мови програмування бувають спеціалізованими: кожною мовою найзручніше вирішуються задачі своєї певної області. якщо ж треба вирішити комплексну задачу, яка розпадається на

частини, що належать до різних областей, то найкраще кожну підзадачу реалізувати окремим модулем, написаним найзручнішою для цієї підзадачі мовою.

Отже, модульність стала одним з найвизначніших винаходів в історії програмування і досі залишається наріжним каменем сучасних технологій програмування. Модульність – це не лише суто технічний прийом, а ще й ключовий елемент культури програмування, вона займає центральне місце в парадигмі і, без перебільшення, має світоглядне значення.

## Розбиття програми на різні файли

Коли потрібно працювати з достатньо великою програмою часто потрібно розбити її на якийсь певні логічні, функціональні частини які можна, а часто й потрібно винести в різні файли. Наприклад  це робиться для того щоб декілька програмістів працювало над одним проектом, відокремити частини функціоналу та інтерфейсів для різних підпроектів і таке інше.

Розглянемо просту програму:

```
#include <stdio.h>
int add(int x, int y){
     return x+y;
}
int main(){
   printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

Тут доцільно розбити програму на 2 частини - з функцією  add та функцією main.cpp

```
// файл 1  add.c
int add(int x, int y){
```

```
    return x+y;
}
// файл 2: main.c
#include <stdio.h>
int main(){
   printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

Однак, якщо ми просто скомпілюємо тепер файл - то програма може не запрацювати:

warning: implicit declaration of function ‘add’ [-Wimplicit-function-declaration]

Повідомлення каже нам, що немає визначення тіла функції add.

Для того, щоб компіляція відбулася без попереджень можна додати включення першого файлу у другий та скомпілювати його.

```
#include <stdio.h>
#include "add.c"
int main(){
   printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

В цьому випадку директива #include "add.c" просто додає код файлу add.c до другого файлу main.c, а отже команда компіляції ( наприклад gcc main.c ) виконає початкову версію файлу.

## Включення файлів

Для включення та використання функціоналу інших файлів в мові С використовується директива #include .

Синтаксис директиви такий: після слова #include йде ім'я файлу (з розширенням), взяте або в подвійні лапки, або в кутові дужки. Іншими словами, директива має одну з двох форм:

1. **# include < filename.h>**
2. **# include "filename.h"**

Перша форма каже транслятору, що файл, ім’я якого вказано в кутових дужках, треба шукати в спеціальній директорії, призначеній для стандартних заголовочних файлів. Саме тому в кутових дужках вказують імена таких заголовочних файлів, як stdio.h , math.h та інших, які поставляються разом з транслятором. Друга форма директиви означає, що транслятор повинен шукати файл в директорії для користувацьких заголовочних файлів (можливо, зокрема, у тій самій директорії, що й текст модуля).

Сенс директиви пояснимо спочатку спрощено. Зустрівши будь-де в тексті директиву include, транслятор має знайти файл з відповідним іменем, та підставити весь вміст цього файлу замість директиви.

Покажемо це на прикладі. Нехай є файл mydecl.h

```
// mydecl.h
typedef unsigned long int UL;
# define MAX_LEN 32
UL functionSample( int * k , UL x );
```

та ще один файл з іменем mycode.c :

```
// mycode.c
# include " mydecl . h " /* Включили файл mydecl.h !!! */

int main () {
int m [ MAX_LEN ];
UL s , r =0;
s = functionSample(m , r );
```

```
. . . /* ще якісь оператори */
}
```

В файлі mycode.c компілятор прочитавши другий рядок знайде mydecl.h та вставить його на місце директиви. Отже, після такої підстановки компілятор «побачить», ніби файл mycode.c виглядає так:

```
typedef unsigned long int UL;
# define MAX_LEN 32
UL functionSample( int * k , UL x );
int main () {
int m [ MAX_LEN ];
UL s , r =0;
s = functionSample( m , r );
. . . /* ще якісь оператори */
}
```

Наведене вище пояснення роботи директиви насправді трохи спрощене. Більш точно процес виглядає так: транслятор читає файл рядок за рядком; коли транслятор бачить в тексті директиву #include , він тимчасово припиняє обробляти поточний файл, шукає той файл, ім'я якого вказано в лапках після слова include, та починає так само, рядок за рядком, опрацьовувати його; дійшовши до кінця заголовочного файлу, транслятор знов повертається до недочитаного файлу та продовжує обробляти його з того самого місця, де припинив.

Однак, це не дуже гарне рішення з точки зору створення саме бібліотеки для компіляції. Бажано створити окремий модуль для роботи з створеними функціями.

Для цього можна створити файл  з використанням прототипів (попередньої декларації):

```
//main.c (з  попередньою декларацією - forward declaration):
```

```
#include <stdio.h>
extern int add(int x, int y); // функція вказуємо що є функція add()
// без специфікатору extern в принципі можна обійтися
int main(){
   printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
   return 0;
}
// файл add.cpp :
int add(int x, int y){
     return x+y;
}
```

Тепер, коли компілятор компілює main.cpp, він буде знати, який ідентифікатор функції додати і бути скомпільованим. Конпоновщик (лінкер) з'єднає виклик функції, щоб додати в main.cpp визначення функції add в add.c. Але для цього, щоб це відбулося, потрібно зкомпілювати обидва файли разом (наприклад, **gcc main.c add.c**).

Зауважимо, що ще краще додати специфікатор extern до функції int add(int x, int y); в файлі main.c.

Крім того, не завжди зовсім зручно додавати прототипи функцій на початку файлу. Крім того, хотілося б компілювати один файл замість того, щоб вказувати багато файлів.

Тоді є наступний варіант - додати #include для реалізації функції:

```
#include <stdio.h>
#include "add.c"
int main(){
   printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

Але це фактично той самий попередній варіант, тут поки що немає бібліотеки з інтерфейсом. На Сі для створення таких багатофайлових застосувань створюють заголовочні файли. Для даної програми це буде тоді виглядати так:

```
//Файл add.c :
int add(int x, int y){
   return x + y;
}
// Файл add.h :
extern int add(int x, int y);
// Файл main.c:
#include <stdio.h>
#include "add.h"
int main(){
   printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
   return 0;
}
```

Компіляція знов іде для обох файлів разом (наприклад, **gcc main.c add.c**).

Використовуючи цей метод, ми можемо надавати файлам доступ до функцій, які створені в іншому файлі.

Спробуйте самі скомпілювати add.c і main.c . Якщо ви отримали помилку лінкера, переконайтеся, що ви додали add.c у вашому проекті чи рядку компіляції належним чином.

### Проблеми, які виникають при компіляції багатьох файлів

Є багато речей, які можуть піти не так, коли ви вперше намагаєтеся працювати з декількома файлами. Якщо ви спробували вказаний вище приклад і виникли помилки, перевірте наступне:

1. Якщо ви отримаєте помилку компілятора про те, що add не визначена в main, ви, напевно, забули прототип для функції add в main.cpp.

2. Якщо ви отримуєте помилку лінкера про не визначену функцію, наприклад, невирішений зовнішній символ "int __cdecl add (int, int)" (? add @@ YAHHH @ Z), на який посилається функція _main, то

2a. - найімовірнішою причиною є те, що add.cpp не додано до вашого проекту правильно. Під час компіляції ви повинні побачити список компіляторів як main.cpp, так і add.cpp. Якщо ви бачите тільки main.cpp, то add.cpp точно не збирається. Якщо ви використовуєте Visual Studio або Code :: Blocks, ви повинні побачити add.cpp у списку Solution Explorer / Project з лівого боку IDE. Якщо ви не хочете, клацніть правою кнопкою миші на своєму проекті та додайте файл, а потім спробуйте його знову Якщо ви компілюєте в командному рядку, не забудьте включити як main.cpp, так і add.cpp у команду компіляції.

2b. - можливо, ви додали add.cpp до неправильного проекту.

2c. - можливо, файл встановлено так, щоб він не компілювався або посилався. Перевірте властивості файлу та переконайтеся, що файл налаштований на компіляцію/лінковку. У Code::Blocks компіляція та посилання є окремими прапорцями, які слід перевірити. У Visual Studio є опція "виключити з побудови", яка повинна бути встановлена на "ні" або залишена порожньою.

3. Не робить #include “add.c” в файлі main.c. Це призведе до того, що компілятор вставить вміст add.c безпосередньо в main.c замість того, щоб розглядати їх як окремі файли. Хоча він може компілюватися і виконуватися для цього простого прикладу, ви будете стикатися з проблемами використовуючи цей метод.

## Висновки

Коли компілятор компілює багатофайлову програму, вона може компілювати файли в будь-якому порядку. Крім того, він компілює кожен файл окремо, не знаючи, що знаходиться в інших файлах.

Реальні проекти завжди працюють з декількома файлами (а часто їх бувають десятки а то і сотні, до того ж з використанням сторонніх бібліотек), тому в

важливо переконатися, що ви розумієте, як додавати і компілювати декілька файлів.

Нагадування: кожен раз, коли ви створюєте новий файл (.c, .cc або .cpp), вам потрібно буде додати його до свого проекту, щоб його було зібрано.

## Модульна організація програм

Щоб добре зрозуміти принцип побудови багатомодульної програми, варто спочатку уявити деяку достатньо велику програму, яка вся міститься в одному файлі. Вона повинна містити декілька функцій, кожна з яких може викликати будь-яку іншу функцію. Оскільки будь-яка функція повинна бути оголошена перед першим викликом, а кожна функція програми може викликати будь-яку іншу функцію, на самому початку програми повинні стояти оголошення усіх функцій (крім main ).

Тепер уявімо, що цю ж програму розбито на кілька модулів, тобто тіла декількох функцій перенесено в один файл, тіла кількох інших функцій – у другий, і т.д. Функція, розміщена в одному модулі, може викликати функцію, що міститься у іншому модулі.

А це означає, що на початку кожного модуля повинні міститися оголошення (прототипи) всіх функцій  функцій з інших модулів, які викликаються з цього модуля.

## Бібліотеки

**Бібліотека** - це пакет коду, який призначений для повторного використання багатьма програмами. Як правило, бібліотека C ++ складається з двох частин:

1) Файл заголовка, який визначає функціональні можливості, які бібліотека виставляє (пропонує) програмам, що використовують її.

2) Скомпільований бінарний файл, який містить реалізацію цієї функціональності, попередньо скомпільовану в машинну мову.

Деякі бібліотеки можуть бути розділені на кілька файлів та/або мати декілька файлів заголовків.

Бібліотека: Бібліотека - це місце, де реалізована фактична функціональність, тобто вони містять тіло функції. Бібліотеки мають переважно дві категорії:

- **статичні бібліотеки,**
- **динамічні бібліотеки.**

Бібліотеки попередньо компілюються з кількох причин. По-перше, оскільки бібліотеки рідко змінюються, їх не потрібно часто перекомпілювати. Було б марно витрачати час на перекомпіляцію бібліотеки кожного разу, коли ви писали програму, яка їх використовувала. По-друге, оскільки попередньо скомпільовані об'єкти знаходяться на машинній мові, це запобігає доступу людей або зміні вихідного коду.

### Різниця між файлом заголовка та бібліотекою

Файли заголовків - це файли, які повідомляють компілятору, як викликати деякі функціональні можливості (не знаючи, як насправді працює функціональність), називаються заголовочними (хедер) файлами. Вони містять прототипи функцій. Вони також містять типи даних і константи, що використовуються в бібліотеках.

На мові Сі використовується #include для використання цих файлів заголовків у програмах. Ці файли закінчуються розширенням .h.

**Приклад**: Math.h - це заголовочний файл, що включає в себе прототип викликів функцій, таких як sqrt (), pow () і т.д. Простими словами, заголовочний файл подібний до візитної картки, а бібліотеки - як справжня людина, тому ми використовуємо візитну картку (файл заголовка), щоб дістатися до фактичної особи (бібліотека).

Давайте побачимо різницю між цими двома в табличній формі, щоб можна було легко порівняти:

Таблиця 6.1

Порівняння заголовочних та бібліотечних файлів

| Файли заголовків | Файли бібліотеки |
|---|---|
| Вони мають розширення .h | Вони мають розширення .lib,.dll, .so, .a |
| Вони містять декларацію функцій. Вони містять визначення функцій | Вони містять визначення функцій |
| Вони доступні всередині підкаталогу проекту | Вони доступні в підкаталогу бібліотек |
| Файли заголовків читаються людиною. Тому що вони мають форму вихідного коду. | Файли бібліотек неможливо читати, оскільки вони мають форму машинного коду. |
| Файли заголовків в нашій програмі включені за допомогою команди #include, яка внутрішньо обробляється попередньою обробкою. | Файли бібліотеки в нашій програмі включені в останню стадію спеціальним програмним забезпеченням, що називається компоновщик(linker) |

## Заголовочні файли

Прототипи зовнішніх функцій, звичайно ж, можна було б вписати в текст модуля вручну, але при великій кількості функцій це було б надто незручно. Набагато зручніше виносити прототипи функцій в окремий заголовочний файл (за допомогою директиви #include ) файл (файл з розширенням .h ) та підключати до кожного модуля, який має намір використовувати ці функції.

Отже, створюючи кожен модуль, програміст в один файл записує тіла кількох функцій в файл що має розширення .c або .cpp(.cc etc), а в інший, заголовочний, з розширенням .h - лише прототипи цих функцій. **Про заголовочний файл часто кажуть, що він містить інтерфейси модуля або прототипи функцій,** а про .c –файл що він є **реалізацією модуля.**

Трансляція багатомодульної програми має важливу особливість. Компілятор обробляє не всю багатомодульну програму як одне ціле, а окремо модуль за модулем. Модулі компілюються незалежно один від одного, іншими словами, компілятор, поки компілює один модуль не знає про існування інших модулів. Наприклад, коли модуль unit1.c викликає наприклад, деяку функцію int func1(

int ) модуля unit2.c, то він в даний момент «бачить» лише інтерфейс функції з unit2.h а не її реалізацію в unit2.c.

Тоді компілятор, поки обробляє модуль використовує ім'я функції, що викликається (воно відомо з прототипу), але не щоб дізнатися тіло цієї функції.

В результаті такої роздільної компіляції кожного с-файлу виходить так званий об'єктний файл, який зазвичай має таке ж ім'я, що й відповідний вхідний файл, але з розширенням .obj для windows-подібних систем або .o для Unix-подібних).

В об'єктному файлі операції над даними та оператори управління вже перекладено у машинні коди, але виклики функцій поки що залишено, як прийнято говорити, наприклад, об'єктний файл unit1.obj містить виклик функції int func1( int ), але не саму функцію

Якщо реалізація цієї функції відсутня, то лінковка є нерозв'язаною оскільки неможливо перетворити на машинну команду переходу до тіла функції, бо модуль не знає нічого про об'єктний файл unit2.obj , в якому міститься це тіло.

Нарешті, після того, як модулі відкомпільовано, спрацьовує ще одна спеціальна програма компоновщик (лінковщик) , або редактор зв'язків.

Спрощено кажучи, він бере всі об'єктні файли, що входять до складу програми, та об'єднує їх в один виконуваний файл (тобто файл, готовий для виконання на машині), розв'язуючи при цьому посилання на імена функцій та замінюючи виклик функції за іменем, що міститься в одному модулі, на конкретну адресу її тіла,взятого з іншого модуля.

**Примітка 1**:  Мова С сама по собі призначена для того, щоб писати нею модулі, і тільки для цього. Опис того, з яких модулів складається багатомодульна програма, здійснюється не за допомогою мови програмування, а спеціальними засобами, наприклад Makefile, скріпт на мовах типу Bash, Python або за допомогою середовища IDE (MS Visual Studio etc).

**Примітка 2:** Стандарт та означення мови С не містять правил щодо імен та розширень заголовочних файлів в принципі в директиві підключення можна вказати ім'я файлу з будь-яким розширенням (головне, щоб цей файл містив текст, зрозумілий для транслятора, тобто текст мовою Сі). Але за усталеною

традицією файлам, спеціально призначеним для включення в інші файли, надають розширення “**.h**” від англ. header (заголовок).

Як вже неодноразово зазначалося вище, в заголовочних файлах прийнято розміщувати оголошення , потрібні в основному тексті програми. Але важливо розуміти, що використання .h – файлів саме для оголошень не синтаксичне правило, а традиція. Транслятор не перевіряє, знаходяться у підключеному файлі оголошення чи якісь інші конструкції мови Сі, єдине, що транслятор вимагає, це щоб у підключеному файлі був деякий текст, правильний з точки зору граматики мови Сі.

Тому, в принципі, правила мови дозволяють і такий трюк, який, на жаль, часто приваблює початківців: розбити довгий програмний текст на кілька частин, кожну оформити в окремому файлі, а в ще одному файлі зібрати ці частини до купи за допомогою директиви include .

Дуже типова помилка початківців полягає в тому, що модульність підміняють таким включенням шматків програмного коду з різних файлів. Насправді це не має нічого спільного з модульністю. Модулі, як вже було сказано вище, повинні компілюватися незалежно один від одного, щоб програма як ціле могла збиратися з цих частин. А в описаному тут помилковому підході всі частини програмного коду компілюються разом завдяки директивам включення компілятор побачить один суцільний програмний текст.

**Увага**!

В заголовочному файлі можна розміщувати: прототипи функцій, оголошення типів (особливо структур), означення макросів і всі такі оголошення, які використовуються на етапі компіляції. В жодному разі не слід розміщувати в заголовочному файлі програмний код, що стосується етапу виконання програми тобто тіла функцій.

**Примітка**. Насправді більшість компіляторів дозволять вам написати тіло функцій в заголовочному файлі. Єдине реальне обмеження - це неможливість написання головної функції main в заголовочному файлі.

### Проблема подвійного включення

Використовуючи викладені вище засоби, можна зіткнутися з серйозними незручностями.

Спочатку опишемо та проілюструємо сутність проблеми, а потім запровадимо зручний засіб для їх усунення.

На практиці можливі ситуації, коли один заголовочний файл прямо чи непрямо підключається до якогось модуля два або більше разів. Це означає, що кожне оголошення з цього заголовочного файлу потрапить в текст програми кілька разів, а це може викликати помилку компіляції. Розглянемо приклад. Нехай в заголовочному файлі один структурний тип:

```
// файл ratio.h
typedef struct tagRatio {
int m , n ;
} TRatio ;
```

Нехай є ще два заголовочні файли, в яких оголошуються функції для роботи з об'єктами TRatio. У файлі ratio_io.h оголосимо функції для введення та виведення а в ratio_m.h функції для математичних обчислень з ними. В кожному цих двох заголовочних файлів першим рядком йде підключення файлу ratio.h :

```
// Файл ratio_io.h :
# include "ratio.h"
TRatio ratio_read ();
void ratio_print ( TRatio );
//Файл ratio_m.h :
# include " ratio . h "
TRatio ratio_add ( TRatio , TRatio );
TRatio ratio_mpy ( TRatio , TRatio );
```

Нарешті, нехай є модуль, який має намір використовувати обидва набори функцій:

```
// Файл main.c
# include " ratio_io . h"
# include " ratio_m . h "
/* якийсь програмний текст */
```

Розберемо, як транслятор прочитає цей текст. Натрапивши на перший рядок, він (спрощено кажучи) вставить на це місце файл є директива включення файлу ratio.h ,ratio_io.h та спробує обробити його, а у цьому файлі тому транслятор підставить вміст і цього файлу. Так само, обробляючи другий рядок, транслятор змушений буде ще раз вставити текст файлу ratio.h .

Отже, з точки зору транстятора модуль після обробки всіх директив виглядатиме так:

```
typedef struct tagRatio {
int m , n ;
} TRatio ;
TRatio ratio_read ();
void ratio_print ( TRatio );
typedef struct tagRatio {
int m , n ;
} TRatio ;
TRatio ratio_add ( TRatio , TRatio );
TRatio ratio_mpy ( TRatio , TRatio );
/* якийсь програмний текст */
```

як видно, при компіляції модуля транслятор побачить два означення структурного типу TRatio - а це є помилкою з точки зору граматики мови С.

Отже, як видно з цього прикладу, помилки можуть виникати через те, що один заголовочний файл підключається до кількох інших заголовочних файлів. Коли ці останні підключаються до якогось модуля, весь текст першого файлу

включається багаторазово. Тому виникає задача: винайти такий спосіб написання та використання заголовочних файлів, який би гарантовано уберігав програміста від проблем з подвійним включенням. Іншими словами, хотілося б знайти такий засіб, який би примушував компілятор автоматично слідкувати за спробами багатократно підключити один заголовочний файл, щоб цей файл підключався лише один раз, при першій такій спробі.

### Умовна компіляція

Мова С містить директиви препроцесора #ifdef та #ifndef , які дозволяють на етапі компіляції в залежності від тієї чи іншої умови вилучити або, навпаки, включити той чи інший фрагмент тексту.

Конструкцію

```
# ifdef ім 'я
/* якийсь програмний текст */
#endif
```

компілятор обробляє так: спочатку він перевіряє, чи означене дане ім'я , тобто чи був десь раніше при компіляції цього ж модуля означений макрос з таким іменем; якщо ні, то весь програмний текст до директиви ігнорується. Директива #ifndef #endif пропускає, що текст не компілюється, а якщо так, то #endif працює навпаки: вилучає з компіляції текст до директиви тоді, коли макрос з вказаним іменем означений.

Ці директиви, які називають директивами умовної компіляції, мають багато різних застосувань, які, однак, далеко виходять за рамки ознайомчого курсу для початківців. Зараз опишемо лише одну конструкцію, яка дозволяє вирішити проблему повторного включення.

Розглянемо файл ratio.h у такій редакції:

```
# ifndef _RATIO_H_
# define _RATIO_H_
typedef struct tagRatio {
int m , n ;
```

```
} TRatio ;
# endif
```

Нехай він, як і раніше, підключається до модуля двічі через посередництво інших заголовочних файлів. При першому включенні компілятор, натрапивши на директиву продовжить компіляцію, оскільки макрос з іменем _RATIO_H_ раніше не означувався. В наступному ж рядку даний макрос стає означеним. Після цього при другому включенні того ж заголовочного файлу компілятор, знов обробляючи директиву _RATIO_H_ ifndef , помітить, що макрос вже означено, отже пропустить весь текст заголовочного файлу. Таким чином, вдалося запобігти повтору раніше відкомпільованих оголошень.

Продемонстрована тут техніка є стандартною для мови С: якщо придивитися до фірмових заголовочних файлів, що входять, наприклад, до комплекту середовища програмування, в них можна побачити той же технічний прийом.

**Примітка 1**. Зауважимо також, що ця методика дозволяє також визначити чи був підключений конкретний модуль в дану програму за допомогою виклику директиви #ifdef _RATIO_H_

**Примітка 2**. Деякі середовища пропонують інший варіант боротьби з повторним включенням - додаванню до модуля директиви #pragma (once), але нажаль такий варіант підтримується не всіма компіляторами.

### Включення Сі-модулів в програму на Сі++

При включенні Сі-модулів у проект, що компілюється Сі-компілятором можуть виникнути деякі проблеми пов'язані з тим, що компілятори Сі та Сі++ трошки по різному компілюють код та, наприклад, інтерпретують прототипи функцій. Але є стандартний спосіб створення заголовочних файлів, що дозволяють їм бути підтриманим на Сі++ компіляторі при цьому коректно працювати на Сі-компіляторах.

Приклад.

**//Файл - sum.h:**

```
#ifdef __cplusplus
    extern "C" {
#endif
int sumI (int a, int b);
float sumF (float a, float b);
#ifdef __cplusplus
} // end extern "C"
#endif
```

Як тут будуть працювати блоки умовної компіляції ifdef / endif? Якщо включити цей заголовок у вихідний файл C, то він стане:

```
    int sumI (int a, int b);
    float sumF (float a, float b);
```

Але якщо включити їх із вихідного файлу C ++, то він стане:

```
    extern "C" {
    int sumI (int a, int b);
    float sumF (float a, float b);
    } // end extern "C"
```

Мова C нічого не знає про директиву extern "C", але C ++ вміє її обробити, і він потребує цієї директиви для оголошень C-функції. Це пов'язано з тим, що C ++ викликає імена функцій (і методів) та він підтримує перевантаження функції або методів, тоді як C не підтримує.

Зокрема це можна побачити у вихідному файлі C ++ з назвою print.cpp:

```
    #include <iostream> // std :: cout, std :: endl
    #include "sum.h" // sumI, sumF
void printSum (int a, int b) {
     std::cout << a << "+" << b << "=" << sumI (a, b) << std :: endl;
    }

    void printSum (float a, float b) {
```

```
    std:: cout << a << "+" << b << "=" << sumF (a, b) << std :: endl;
}
  extern "C" void printSumInt (int a, int b) {
  printSum (a, b);
}
  extern "C" void printSumFloat (float a, float b) {
  printSum (a, b);
}
```

## Об'єктні файли

Кожен вихідний файл C та C ++ потрібно компілювати в об'єктний файл. Об'єктні файли, отримані в результаті компіляції декількох вихідних файлів, потім пов'язуються з виконуваним файлом, спільною бібліотекою або статичною бібліотекою (останній з них є лише архівом об'єктних файлів). Файли C ++ зазвичай мають суфікси розширення .cpp, .cxx або .cc.

Вихідний файл C ++ може включати інші файли, відомі як файли заголовків, з директивою #include. Файли заголовків мають розширення, такі як .h, .hpp або .hxx, або взагалі не мають розширення, як у типовій бібліотеці C ++ та інших заголовках бібліотек. Розширення не має значення для препроцесора C ++, який буквально замінить рядок, що містить директиву #include, на весь вміст включеного файлу.

Першим кроком, який компілятор зробить у вихідному файлі, буде запустити препроцесор на ньому. Тільки вихідні файли передаються компілятору (для попередньої обробки та компіляції). Файли заголовків не передаються компілятору. Натомість вони включені з вихідних файлів.

Кожен заголовочний файл може бути відкритий кілька разів під час фази попередньої обробки всіх вихідних файлів, залежно від того, скільки вихідних файлів містять їх, або скільки інших заголовних файлів, включених з вихідних

файлів, також включати їх (може бути багато рівнів непрямого) . З іншого боку, вихідні файли відкриваються тільки один раз компілятором (і препроцесором), коли вони передаються йому.

Для кожного вихідного файлу C ++ препроцесор побудує модуль трансляції, вставивши в нього вміст, коли одночасно знайде директиву #include, яку він буде видаляти з вихідного файлу коду та заголовків, поки не знайде умовну компіляцію блоки, директиви яких оцінюються як помилкові. Також будуть виконуватися інші задачі, наприклад, заміна макросів.

Після того, як препроцесор завершить створення цієї (іноді величезної) одиниці трансляції, компілятор починає фазу компіляції і створює об'єктний файл.

*Щоб отримати цю одиницю перекладу (попередньо оброблений вихідний код), аргумент (опцію) -E можна передати компілятору g ++ разом з опцією -o, щоб вказати бажане ім'я попередньо обробленого вихідного файлу.*

### Статичні та динамічні бібліотеки

Існує два типи бібліотек: статичні бібліотеки та динамічні бібліотеки.

**Статичні бібліотеки** містять об'єктний код, пов'язаний з додатком кінцевого користувача, а потім вони стають частиною виконуваного файлу. Ці бібліотеки спеціально використовуються під час компіляції, що означає те, що бібліотека повинна бути у правильному місці, коли користувач хоче скомпілювати свою програму C або C ++. У Windows вони закінчуються розширенням .lib, а для для Unix-подібних систем та MacOS - вони мають розширення .a.

Статична бібліотека (також відома як архів) складається з процедур, які компілюються і безпосередньо пов'язані з вашою програмою. Під час компіляції програми, яка використовує статичну бібліотеку, вся функціональність статичної бібліотеки, що використовує наша програма, стає частиною виконуваної програми. У Windows статичні бібліотеки зазвичай мають розширення .lib, тоді як у Unix-подібних систем статичні бібліотеки зазвичай

мають розширення .a (архіву). Однією з переваг статичних бібліотек є те, що потрібно завантажувати або розповсюджувати лише виконуваний файл, щоб користувачі могли запускати вашу програму. Оскільки бібліотека стає частиною вашої програми, це гарантує, що з вашою програмою завжди використовується правильна версія бібліотеки. Крім того, оскільки статичні бібліотеки стають частиною вашої програми, ви можете використовувати їх так само, як і функціональні можливості, які ви написали для своєї програми. З іншого боку, оскільки копія бібліотеки стає частиною кожного виконуваного файлу, це може призвести до великої кількості втраченого простору. Статичні бібліотеки також не можуть бути легко оновлені - для оновлення бібліотеки необхідно замінити весь виконуваний файл.

**Динамічна бібліотека** (яка також називається спільною бібліотекою, shared library) складається з процедур, які завантажуються у вашу програму під час виконання. Коли ви збираєте програму, яка використовує динамічну бібліотеку, бібліотека не стає частиною вашого виконуваного файлу - вона залишається окремою одиницею. У Windows динамічні бібліотеки, як правило, мають розширення .dll (динамічна бібліотека зв'язків), тоді як у Linux динамічні бібліотеки зазвичай мають розширення .so (shared object). Однією з переваг динамічних бібліотек є те, що багато програм можуть спільно використовувати одну копію, що економить простір. Можливо, більшою перевагою є те, що динамічну бібліотеку можна оновити до більш нової версії без заміни всіх виконуваних файлів, які використовують її.

Спільні або динамічні бібліотеки потрібні лише під час виконання, тобто користувач може компілювати свій код без використання цих бібліотек. Коротше кажучи, ці бібліотеки пов'язані між собою під час компіляції для вирішення невизначених посилань, а потім поширюються до програми, щоб програма могла завантажувати її під час виконання. Наприклад, коли ми відкриваємо наші папки з іграми, можна знайти багато файлів .dll (динамічних бібліотек). Оскільки ці бібліотеки можуть спільно використовуватися кількома

програмами, вони також називаються спільними бібліотеками. Ці файли закінчуються розширеннями .dll або .lib.

Оскільки динамічні бібліотеки не пов'язані з вашою програмою, програми, що використовують динамічні бібліотеки, повинні явно завантажувати та взаємодіяти з динамічною бібліотекою. Цей механізм може бути заплутаним і робить незручною взаємодію з динамічною бібліотекою. Для спрощення використання динамічних бібліотек можна використовувати бібліотеку імпорту.

**Бібліотека імпорту** - це бібліотека, яка автоматизує процес завантаження та використання динамічної бібліотеки. У Windows, це зазвичай робиться через невелику статичну бібліотеку (.lib) з тією ж назвою, що й динамічна бібліотека (.dll). Статична бібліотека пов'язана з програмою під час компіляції, а потім функціональність динамічної бібліотеки може бути ефективно використана, як якщо б вона була статичною бібліотекою. У Linux файл спільного об'єкта (.so) використовується і як динамічна бібліотека, так і як бібліотека імпорту. Більшість компоновщиків може створювати бібліотеку імпорту для динамічної бібліотеки, коли створюється динамічна бібліотека.

## Встановлення та використання бібліотек

Процес, необхідний для використання бібліотеки:

Для кожної бібліотеки:

1) Закачайте бібліотеку. Наприклад, завантажте з веб-сайту або за допомогою менеджера пакетів.

2) Встановіть бібліотеку. Розпакуйте її до каталогу або встановіть за допомогою менеджера пакетів, або проінсталюйте її (наприклад, запустить як C/C++ - програму).

3) Вкажіть компілятору, де слід шукати файл(и) та заголовок(ки) для бібліотеки.

Для кожного проекту:

4) Вкажіть лінкеру (компонувальнику), де шукати файл(и) бібліотеки для бібліотеки.

5) Вкажіть лінкеру (компонувальнику), які статичні файли або файли імпортувати.

6) Включіть файли заголовків бібліотеки у вашу програму.

7) Переконайтеся, що програма знає, де можна знайти будь-які динамічні бібліотеки, які використовуються.

Те саме більш докладно:

Встановлення бібліотеки на C ++ зазвичай включає 4 кроки:

1) Завантаження бібліотеки. Найкращим варіантом є завантаження попередньо скомпільованого пакета для вашої операційної системи (якщо він існує), тому вам не доведеться самостійно компілювати бібліотеку. Якщо для вашої операційної системи немає такого пакету, вам доведеться завантажити доданок з вихідним кодом і скомпілювати його самостійно (компіляція проекту).

У Windows бібліотеки, як правило, розповсюджуються у вигляді файлів .zip або безпосередньо dll. У Linux бібліотеки зазвичай розподіляються як пакети (наприклад, .RPM) або теж архівом. Менеджер пакетів може мати деякі з найпопулярніших бібліотек (наприклад, SDL), перероблені вже для легкого встановлення, тому перевірте їх там і встановить, наприклад як apt install.

2) Встановіть бібліотеку. У Linux це зазвичай передбачає виклик менеджера пакетів і дозвіл (наприклад надання системних прав) йому виконати всю роботу. У Windows це зазвичай передбачає розпакування бібліотеки в потрібний каталог. Ми рекомендуємо зберігати всі ваші бібліотеки в одному місці для легкого доступу. Наприклад, використовуйте каталог з назвою C:\\Libs і помістіть кожну бібліотеку в її власний підкаталог.

3) Переконайтеся, що компілятор знає, де шукати файли заголовків для бібліотеки. У Windows, як правило, це підкаталог include каталогу, до якого ви

встановили файли бібліотеки (наприклад, якщо ви встановили бібліотеку до C:\\libs SDL-1.2.11, файли заголовків, ймовірно, знаходяться в C:\\libs, а -1.2.11 виключають з назви). На Linux бібліотеки зазвичай встановлюються в /usr/include, які вже повинні бути частиною шляху пошуку файлів. Однак, якщо файли встановлені в іншому місці, вам доведеться сказати компілятору, де їх знайти.

4) Вкажіть компонувальнику, де шукати файли бібліотек. Як і в кроці 3, це зазвичай передбачає додавання каталогу до списку місць, де лінкер шукає бібліотеки, або додайте бібліотеки в ті місця де проект вже шукає бібліотеки (зокрема в системних шляхах) . У Windows, це типово підкаталог /lib каталогу, до якого ви встановили файли бібліотеки. В Linux бібліотеки зазвичай встановлюються в /usr/lib, які вже повинні бути частиною шляху пошуку вашої бібліотеки.

Після того, як бібліотека встановлена та IDE знає, де її слід шукати, для кожного проекту, який бажає використовувати бібліотеку, як правило, потрібно виконати наступні 3 кроки:

5) Якщо ви використовуєте статичні бібліотеки або імпортовані бібліотеки, повідомте компонувальнику, які файли бібліотек повинні зв'язати (gcc -L(l) *).

6) Включіть файли заголовків бібліотеки у вашу програму (директива include). Це повідомляє компілятору про всі функціональні можливості, які пропонує бібліотека, щоб програма могла правильно скомпілюватись.

7) Якщо використовуються динамічні бібліотеки, переконайтеся, що програма знає, де їх можна знайти. У Linux, бібліотеки, як правило, встановлюються в /usr/lib, який знаходиться в шляху пошуку за замовчуванням або каталоги змінної середовища PATH. У Windows шлях пошуку за замовченням включає в себе каталог, з якого запускається програма, каталоги, встановлені за допомогою виклику SetDllDirectory(), каталоги Windows, System і System32, а також каталоги змінної середовища PATH. Найпростіший спосіб використовувати .dll - скопіювати .dll в розташування виконуваного файлу.

Оскільки ви зазвичай розповсюджуєте файл .dll із вашим виконуваним файлом, має сенс тримати їх разом.

Кроки 3-5 включають налаштування вашого IDE - на щастя, майже всі IDE працюють так само, коли йдеться про виконання цих речей. На жаль, оскільки кожна IDE має інший інтерфейс, найскладнішою частиною цього процесу є просто визначення місця, де потрібне місце для виконання кожного з цих кроків. Ми розглянемо, як виконувати всі ці дії для Visual Studio Express 2005 і Code :: Blocks. Якщо ви використовуєте іншу IDE, прочитайте обидва - до того часу, коли ви закінчите, ви повинні мати достатньо інформації, щоб зробити те ж саме з вашим власним середовищем розробки (IDE) з невеликим пошуком.

### Visual Studio Express 2005

Кроки 1 і 2 - Отримання та встановлення бібліотеки

Завантажте та встановіть бібліотеку на жорсткий диск.

Кроки 3 і 4 - Скажіть компілятору, де можна знайти заголовки та файли бібліотеки

Ми будемо робити це на глобальній основі, тому бібліотека буде доступна для всіх наших проектів. Отже, наступні кроки потрібно робити лише один раз на бібліотеку.

A) Перейдіть до меню «Інструменти»("Tools") та виберіть "Опції"("Options")

B) Відкрийте вкладку "Проекти та рішення" ("Projects and Solutions") і натисніть "Директорії VC ++" ("VC++ Directories").

C) У верхньому правому куті в розділі "Показати каталоги для:" ("Show directories for") виберіть "Включити файли"("Include Files"). Додайте шлях до файлів .h для бібліотеки.

D) У верхньому правому куті в розділі "Показати каталоги для:" ("Show directories for") виберіть "Бібліотечні файли"("Library Files"). Додайте шлях до файлів .lib для бібліотеки.

E) Натисніть "OK".

Крок 5 - Вкажіть компонувальнику, які бібліотеки використовує ваша програма

Для кроку 5 нам потрібно додати .lib файли з бібліотеки до нашого проекту. Ми робимо це на індивідуальній основі. Visual Studio пропонує 3 різні методи для додавання .lib файлів до нашого проекту:

A) Використовуйте директиву препроцесора #pragma до основного файлу .cpp. Це рішення працює лише з Visual Studio і не є портативним. Інші компілятори ігноруватимуть цей рядок.

```
#include "curses.h"
#pragma comment(lib, "PDCurses.lib")
```

B) Додайте файл .lib до вашого проекту, як якщо б він був файлом .cpp або .h. Це рішення працює з Visual Studio, але не з багатьма іншими компіляторами.

B) Додайте бібліотеку на вхід лінкера. Це найбільш «портативне» рішення в тому сенсі, що кожен IDE забезпечить подібний механізм. Якщо ви коли-небудь переходите до іншого компілятора або IDE, це рішення вам доведеться використовувати. Це рішення потребує 5 кроків:

C-1) В вкладці рішень клацніть правою кнопкою миші на назвою проекту з напівжирним шрифтом і виберіть у меню пункт «Властивості»("Properties").

C-2) У спадному меню "Конфігурація:"("Configuration:") виберіть "Всі конфігурації" ("All Configurations").

C-3) Відкрийте вузол “Властивості конфігурації”("Configuration Properties"), вузол “Linker” і натисніть “Input”.

C-4) У розділі "Додаткові залежності"("Additional Dependencies") додайте назву бібліотеки.

C-5) Натисніть "OK".

Кроки 6 і 7 - включають файли заголовків і переконайтеся, що проект може знайти DLL . Просто включіть заголовок (файли) з бібліотеки у ваш проект.

## Середовище Code :: Blocks

Кроки 1 і 2 - Отримання та встановлення бібліотеки

Завантажте та встановіть бібліотеку на жорсткий диск.

Кроки 3 і 4 - Скажіть компілятору, де можна знайти заголовки та файли бібліотеки.

Ми будемо робити це на глобальній основі, тому бібліотека буде доступна для всіх наших проектів. Отже, наступні кроки потрібно робити лише один раз на бібліотеку.

A) Перейдіть до меню «Налаштування» та виберіть «Компілятор».

B) Перейдіть на вкладку "Каталоги". Вкладку компілятора вже буде вибрано для вас.

C) Натисніть кнопку «Додати» і додайте шлях до файлів .h для бібліотеки. Якщо ви працюєте з Linux і встановили бібліотеку за допомогою менеджера пакетів, переконайтеся, що тут є  /usr/include

D) Перейдіть на вкладку "Linker". Натисніть кнопку "Додати" і додайте шлях до файлів .lib для бібліотеки. Якщо ви працюєте з Linux і встановили бібліотеку за допомогою менеджера пакетів, переконайтеся, що /usr/lib вказаний тут.

E) Натисніть кнопку "OK".

Крок 5 - Скажіть компонувальнику, які бібліотеки використовує ваша програма. Ми повинні додати файли бібліотеки з бібліотеки до нашого проекту. Ми робимо це на індивідуальній основі для кожного проекту.

A) Клацніть правою кнопкою миші на назву напівжирного проекту під робочим місцем за умовчанням (можливо, "Консольне додаток", якщо ви його не змінили). Виберіть у меню пункт "Параметри побудови".

B) Перейдіть на вкладку лінкера. У вікні «Бібліотеки посилань» натисніть кнопку «Додати» та додайте бібліотеку, яку ви бажаєте використовувати у вашому проекті.

C) Натисніть кнопку "OK"

Кроки 6 і 7 -  включають файли заголовків і переконайтеся, що проект може знайти DLL. Просто включіть заголовочні файли з бібліотеки у ваш проект.

## *Компілювання*

Зазвичай програми на мовах Сі або Сі ++ є сукупністю кількох .c (.cpp) файлів з реалізаціями функцій і .h файлів з прототипами функцій і визначеннями типів даних. Як правило, кожному .c файлу відповідає .h файл з тим же ім'ям.

Припустимо, що розробляється програма називається general і складається з файлів gen_file1.c, gen_file1.h, gen_file2.c, gen_file2.h, gen_file3.c, gen_file3.h. Розробка програми ведеться в POSIX-середовищі з використанням компілятора GCC.

Найпростіший спосіб скомпілювати програму - вказати всі вихідні .c файли в командному рядку gcc:

```
gcc gen_file1.c gen_file2.c gen_file3.c -o general
```

Компілятор gcc виконає всі етапи компіляції вихідних файлів програми і компоновку виконуваного файлу general. Зверніть увагу, що в командному рядку gcc вказуються тільки .c файли і ніколи не вказуються .h файли.

Компіляція і компоновка за допомогою перерахування всіх вихідних файлів в аргументах командного рядка GCC допустима лише для зовсім простих програм. З ростом числа вихідних файлів ситуація дуже швидко стає некерованою. Крім того, кожен раз все вихідні файли будуть компілюватися від початку до кінця, що в разі великих проектів займає багато часу. Тому зазвичай компіляція програми виконується в два етапи: компіляція об'єктних файлів і компоновка виконуваної програми з об'єктних файлів. Кожному .c файлу тепер відповідає об'єктний файл, ім'я якого в POSIX-системах має суфікс .o. Таким чином, в даному випадку програма general компонується з об'єктних файлів gen_file1.o, gen_file2.o і gen_file3.o наступною командою:

```
gcc gen_file1.o gen_file2.o gen_file3.o -o general
```

Кожен об'єктний файл повинен бути отриманий з відповідного вихідного файлу за допомогою такої команди:

```
gcc -c gen_file1.c
```

Зверніть увагу, що явно задавати ім'я вихідного файлу необов'язково. Воно буде отримано з імені компільованого файлу заміною суфікса .c на суфікс .o. Отже, для компіляції програми general тепер необхідно виконати чотири команди:

```
gcc -c gen_file1.c
gcc -c gen_file2.c
gcc -c gen_file3.c
gcc gen_file1.o gen_file2.o gen_file3.o -o general
```

Хоча тепер для компіляції програми необхідно виконати чотири команди замість однієї, натомість з'являються такі переваги:

• якщо зміна внесено в один файл, наприклад, в файл gen_file3.c, немає необхідності перекомпілювати файли gen_file2.o або gen_file1.o; досить перекомпілювати файл gen_file3.o, а потім виконати компоновку програми general;

• компіляція об'єктних файлів gen_file1.o, gen_file2.o і gen_file3.o не залежить один від одного, тому може виконуватися паралельно на многопроцессорном (багатоядерному) комп'ютері.

У разі декількох вихідних .c і .h файлів і відповідних проміжних .o файлів відстежувати, який файл потребує перекомпіляції, стає складно, і тут на допомогу приходить програма make. За описом файлів і команд для компіляції програма make визначає, які файли потребують перекомпіляції, і може виконувати перекомпіляцію незалежних файлів паралельно.

Файл A залежить від файлу B, якщо для отримання файла A необхідно виконати деяку команду над файлом B. Можна сказати, що в програмі існує залежність файла A від файлу B. У нашому випадку файл gen_file1.o залежить від файлу gen_file1.c, а файл general залежить від файлів gen_file1.o, gen_file2.o

і gen_file3.o. Можна сказати, що файл general транзитивній залежить від файлу gen_file1.c. Залежність файла A від файлу B називається задоволеною, якщо:

• всі залежності файлу B від інших файлів існують;

• файл A існує в файлової системі;

• файл A має дату останньої модифікації не раніше дати останньої модифікації файлу B.

Якщо все залежності файла A існують, то файл A не потребує перекомпіляції. В іншому випадку спочатку задовольняються всі залежності файлу B, а потім виконується команда перекомпіляції файлу A.

Наприклад, якщо програма general компілюється в перший раз, то в файлової системі не існує ні файлу general, ні об'єктних файлів gen_file1.o, gen_file2.o, gen_file3.o. Це означає, що залежно файлу general від об'єктних файлів, а також залежно об'єктних файлів від .c файлів не існують, то є всі вони повинні бути перекомпіліьовані. В результаті в файлової системі з'являться файли gen_file1.o, gen_file2.o, gen_file3.o, дати останньої модифікації яких будуть більше дат останньої модифікації відповідних .c файлів (в припущенні, що годинник на комп'ютері йдє правильно, і що в файлової системі немає файлів "з майбутнього"). Потім буде створено файл general, дата останньої модифікації якого буде більше дати останньої модифікації об'єктних файлів.

Якщо конфігурації всіх залежностей всіх файлів один від одного задоволені, то для компіляції програми general не потрібно виконувати жодних команд

Припустимо тепер, що в процесі розробки був змінений файл gen_file3.c. Його час останньої зміни тепер більше часу останньої зміни файлу gen_file3.o. Залежність gen_file3.o від gen_file3.c стає незадоволеною, і, як наслідок, залежність general від gen_file3.o також стає незадоволеною. Щоб задовольнити залежності необхідно перекомпілювати файл gen_file3.o, а потім файл general. Файли gen_file1.o і gen_file2.o можна не чіпати, так як залежності цих файлів від відповідних .c файлів задоволені. Така загальна ідея роботи програми make і, насправді, всіх програм управління складанням проекту: ant (http://ant.apache.org/), scons (http://www.scons.org/) і ін.

Хоча утиліта make присутня у всіх системах програмування, вигляд керуючого файлу або набір опцій командного рядка можуть сильно відрізнятися. Далі буде розглядатися командна мова і опції командного рядка програми GNU make. У дистрибутивах операційної системи Linux програма називається make. У BSD, як правило, програма GNU make доступна під ім'ям gmake.

Файл опису проекту може містити описи змінних, опису залежностей і опису команд, які використовуються для компіляції. Кожен елемент файлу опису проекту повинен, як правило, розташовуватися на окремому рядку. Для розміщення елементу опису проекту на кількох рядках використовується символ продовження \ точно так же, як в директивах препроцесора мови Сі.

Визначення змінних записуються наступним чином:

**<Ім'я> = <визначення>**

Використання змінної записується в одній з двох форм:

**$ (<Ім'я>)**

або

**$ {<ім'я>}**

Ці форми рівнозначні.

Змінні розглядаються як макроси, тобто використання змінної означає підстановку тексту з визначення змінної в точку використання. Якщо при визначенні змінної були використані інші змінні, підстановка їх значень відбувається при використанні змінної (знову так само, як і в препроцесорів мови Сі)

Залежності між компонентами визначаються наступним чином:

**<Мета>: <мета1> < мета 2> ... < мета N>,**

де <мета> - ім'я мети, яке може бути або ім'ям файлу, або деяким ім'ям, що позначає дію, якому не відповідає ніякої файл, наприклад clean. Список цілей в правій частині задає цілі, від яких залежить <мета>.

Якщо опис проекту містить циклічну залежність, тобто, наприклад, файл A залежить від файлу B, а файл B залежить від файлу A, такий опис проекту є помилковим.

Команди для перекомпіляції мети записуються після опису залежності. Кожна команда повинна починатися з символу табуляції (\ t). Якщо жодної команди для перекомпіляції мети не задано, будуть використовуватися стандартні правила, якщо такі є. Для визначення, яким стандартним правилом необхідно скористатися, зазвичай використовуються суфікси імен файлів. Якщо жодна команда для перекомпіляції мети не задана і стандартне правило, не знайдено, програма make завершується з помилкою.

Для програми general найпростіший приклад файлу Makefile для компіляції проекту може мати вигляд:

```
general: gen_file1.o gen_file2.o gen_file3.o
    gcc gen_file1.o gen_file2.o gen_file3.o -o general

gen_file1.o: gen_file1.c
    gcc -c gen_file1.c

gen_file2.o: gen_file2.c
    gcc -c gen_file2.c

gen_file3.o: gen_file3.c
    gcc -c gen_file3.c
```

Однак, в цьому описі залежностей не враховані .h файли. Наприклад, якщо файл gen_file1.h підключається в файлах gen_file1.c и gen_file2.c, то зміна файлу gen_file1.h повинна приводити до перекомпіляції як gen_file1.c, так і gen_file2.c. Тобто .o файли залежать не лише від .c файлів, але й від .h файлов, які включаються даними .c файлами безпосередньо або опосередковано. З урахуванням цього файл Makefile може отримати наступний вигляд:

```
general: gen_file1.o gen_file2.o gen_file3.o
    gcc gen_file1.o gen_file2.o gen_file3.o -o general

gen_file1.o: gen_file1.c gen_file1.h
    gcc -c gen_file1.c

gen_file2.o: gen_file2.c gen_file2.h gen_file1.h
    gcc -c gen_file2.c

gen_file3.o: gen_file3.c gen_file3.h gen_file1.h
    gcc -c gen_file3.c
```

З урахуванням цих доповнень файл Makefile прийме вигляд:

```
all: general

general: gen_file1.o gen_file2.o gen_file3.o
    gcc gen_file1.o gen_file2.o gen_file3.o -o general

gen_file1.o: gen_file1.c gen_file1.h
    gcc -c gen_file1.c

gen_file2.o: gen_file2.c gen_file2.h gen_file1.h
    gcc -c gen_file2.c

gen_file3.o: gen_file3.c gen_file3.h gen_file1.h
    gcc -c gen_file3.c

clean:
    rm -f general *.o
```

Отримаємо наступний файл:

```
CC = gcc
CFLAGS = -Wall -O2
LDFLAGS = -s

all: general

general: gen_file1.o gen_file2.o gen_file3.o
    $(CC) $(LDFLAGS) gen_file1.o gen_file2.o gen_file3.o -o general

gen_file1.o: gen_file1.c gen_file1.h
    $(CC) $(CFLAGS) -c gen_file1.c

gen_file2.o: gen_file2.c gen_file2.h gen_file1.h
    $(CC) $(CFLAGS) -c gen_file2.c

gen_file3.o: gen_file3.c gen_file3.h gen_file1.h
    $(CC) $(CFLAGS) -c gen_file3.c

clean:
    rm -f general *.o
```

## CMake - програма для створення makefile-ів

CMake одна з найпопулярніших (де-факто стандарт) кросплатформених програм для збирання проектів на Ci та Ci++. CMake не провадить збірку проекту, він лише генерує зі свого файлу збірки make-файл для конкретної платформи. Це може бути проект Microsoft Visual Studio, NMake makefile,

проект XCode для MacOS, Unix makefile, Watcom makefile, проект Eclipse або CodeBlocks.

Даний файл зазвичай називається CMakeLists.txt і містить команди зрозумілі CMake. Виконавши

```
cmake CMakeLists.txt
```

ми отримаємо або проект який можна відкрити у вашому середовищі розробки, або makefile за яким можна зібрати проект запустивши відповідну систему збирання (make, nmake, wmake). Завжди можна явно вказати, що повинна генерувати програма, вказавши ключ -G з потрібним параметром (просто запустіть cmake, щоб побачити доступні варіанти).

Дистрибутив CMake можна скачати з офіційного сайту (http://cmake.org/cmake/resources/software.html) де є версії для всіх популярних платформ або встановити зі сховищ вашого дистрибутива Linux, швидше за все він там уже є.

```
cmake_minimum_required (VERSION 2.6)

set (PROJECT hello_world)
project (${PROJECT})
set (HEADERS hello.h)
set (SOURCES hello.cpp main.cpp)
add_executable ($ {PROJECT} $ {HEADERS} $ {SOURCES})
```

У першому рядку просто вказана мінімальна версія CMake необхідна для успішної інтерпретації файлу.

У третьому рядку визначається змінна PROJECT і їй задається значення hello_world - так буде називатися наша програма.

Про що і йдеться в п'ятому рядку. Конструкція ${ім'я_змінної} повертає значення змінної, таким чином проект буде називатися hello_world.

У сьомому і десятому рядках вводяться змінні містять список файлів необхідних для складання проекту.

І в останньому рядку йде команда зібрати виконуваний файл з ім'ям зазначеному в змінної PROJECT і з файлів імена яких знаходяться в змінних HEADERS і SOURCES.

Цей шаблон CMake зручно використовувати де завгодно, для цього достатньо змінити назву проекту і імена файлів.

### Зборка бібліотеки з CMake

Для збирання бібліотеки потрібно виконати такі самі команди як і виконуваному файлі, але в останньому рядку замість команди add_executable, вказують команду add_library. У цьому випадку буде зібрана статична бібліотека, для складання динамічної бібліотеки треба вказати параметр SHARED після імені бібліотеки:

```
add_library ($ {PROJECT} SHARED $ {HEADERS} $ {SOURCES})
```

**Стильові вимоги**

Правила написання CMake-файлів досить лояльні:

- Імена змінних пишуться англійськими літерами у верхньому регістрі
- Команди CMake записуються англійськими літерами в нижньому регістрі
- Параметри команд CMake пишуться англійськими літерами у верхньому регістрі
- Команди CMake відокремлені пропуском від відкриває дужки

Розташування CMakeLists.txt

Дуже зручно в корені кожного проекту мати директорію build з файлами збірки проекту, причому ім'я файлу у всіх директоріях має називатися однаково (назва за замовчуванням CMakeLists.txt відмінно підійде). Це дозволить збирати складні проекти рекурсивно підключаючи директорії з підпроектів.

Створювати окрему директорію, а не зберігати файл в корені зручно, як з міркувань чітко організованої структури проекту, так і тому, що CMake при роботі створює ряд файлів, які зручно зберігати окремо, щоб не захаращувати проект.

### Збірка складного проекту

Ускладнити завдання і зберемо проект, що залежить від бібліотеки. Для цього треба зробити наступні речі:

- Зібрати бібліотеку
- Підключити бібліотеку до проекту

З пунктом 1 складнощів виникнути не повинно, про те як зібрати виконуваний файл або бібліотеку написано вище.

Тепер про те як цю бібліотеку підключити. Зробити це можна додавши в CMakeLists.txt проекту три рядки:

```
include_directories (../)
add_subdirectory (../ІМЯ_БІБЛІОТЕКІ/build bin / ІМЯ_БІБЛІОТЕКІ)
target_link_libraries ($ {PROJECT} ІМЯ_БІБЛІОТЕКІ)
```

Перший рядок потрібен для того, щоб вказати шлях (в даному випадку це корінь проекту) по якому компілятор буде шукати спільні заголовки.

У другому рядку йде вказівка CMake взяти файл з директорії build підпроекту, виконати його і результат роботи покласти в директорію ./bin/ІМЯ_БІБЛІОТЕКІ.

Третій рядок повинна бути в файлі після команди add_executable і вказує, що проект треба лінковані разом із зазначеною бібліотекою.

Можна додавати ці команди вручну, але можна змінити шаблон, щоб було можна змінювати тільки один рядок, вказуючи ім'я бібліотеки:

```
cmake_minimum_required (VERSION 2.6)
set (PROJECT ІМЯ_ПРОЕКТА)
project ($ {PROJECT})
```

```
include_directories (../)
set (LIBRARIES ІМ'Я_БІБЛІОТЕКІ_1 ІМ'Я_БІБЛІОТЕКІ_2 ***)

foreach (LIBRARY $ {LIBRARIES})
    add_subdirectory (../${LIBRARY}/build bin / $ {LIBRARY})
endforeach ()

set (HEADERS ../ЗАГОЛОВОЧНІ_ФАЙЛИ)
set (SOURCES ../ФАЙЛИ_С)
add_executable ($ {PROJECT} $ {HEADERS} $ {SOURCES})
target_link_libraries ($ {PROJECT} $ {LIBRARIES})
```

**Примітка**. Якщо проекті Microsoft Visual Studio не показані заголовки то потрібно додати в файл два рядки:

```
source_group ( "Header Files" FILES $ {HEADERS})
source_group ( "Source Files" FILES $ {SOURCES})
```

На роботу CMake при генерації файли для інших систем це ніяк не вплине.

**Примітка**

Попередження компілятора можна включити в такий спосіб (приклад для MS VS і GCC):

```
if (MSVC)
    add_definitions (/ W4)
elseif (CMAKE_COMPILER_IS_GNUCXX)
    add_definitions (-Wall -pedantic)
else ()
    message ( "Unknown compiler")
endif ()
```