Piotr Szwed
pszwed@agh.edu.pl

Wykład C –uzupełnienie

Preprocesor

Fazy translacji

Jednostką translacji nazywany jest plik źródłowy (w C lub C++) razem z plikami włączonymi w wyniku działania dyrektywy **`#include`**. Z jednostki translacji mogą być wyłączone fragmenty kodu źródłowego kompilowane warunkowo (dyrektywy **`#if`**).

Jednostki translacji mogą być kompilowane niezależnie. Zazwyczaj kompilowane są tylko te, które zostały zmodyfikowane i ich pliki wynikowe (OBJ) są starsze niż pliki źródłowe lub pliki włączane za pomocą dyrektywy **`#include`**.

Fazy translacji

- Wstępna translacja znaków
  W tej fazie następuje translacja znaków w pliku źródłowym do wewnętrznej reprezentacji używanej przez kompilator. W języku C/C++ ogranicza się to do translacji tzw. *trigraphs* (trzyznakowych ciągów typu `'??!' '??>'`.
- Konkatenacja linii
  Wszystkie linie kodu zakończone znakiem \ są łączone z następną linią. Na przykład:

```
void ma\
in()
{
}
```

- Podział na symbole
  Następuje wydzielenie symbolów i białych znaków. Komentarze są zastępowane pojedynczą spacją. Zachowywane są znaki przejścia do nowej linii.
- Faza preprocesora
  Wykonywane są dyrektywy preprocesora (rozwijanie makr, kompilacja warunkowa). Dla każdego pliku włączanego za pomocą dyrektywy **`#include`** wykonuje się wcześniejsze kroki.
- Odwzorowanie znaków

Znaki oraz sekwencje *escape* są odwzorowywane w odpowiednie wartości należące do zestawu znaków ASCII.

- Konkatencja łańcuchów znaków
  Wszystkie bezpośrednio sąsiadujące stałe tekstowe są sklejane ze sobą.
- Właściwa translacja
  Wydzielone symbole są analizowane pod względem składni. Budowany jest kod plików wynikowych OBJ.
- Konsolidacja
  Pliki OBJ są łączone ze sobą i z kodem zawartym w bibliotekach. Uzgadniane są wzajemne wywołania pomiędzy modułami translacji. Powstaje plik wykonywalny EXE (lub biblioteka).

Dyrektywy preprocesora

Dyrektywy preprocesora (takie jak **`#define`** **`#ifdef`**) są stosowane, aby:

- ułatwić modyfikację kodu źródłowego i jego rekompilację w innym środowisku
- poprawić czytelność kodu

Dyrektywy preprocesora umożliwiają:

- zastępowanie symbolów w tekście programu innymi symbolami
  ```
  #define SIZE 32
  char buf[SIZE];
  for(i=0;i<SIZE;i++){...}
  ```
- wyłączanie fragmentów kodu z kompilacji
  ```
  #ifdef _DEBUG
  printf("x=%d",x);
  #endif
  ```

- włączanie plików
  `#include <xxx.h>`

Składnia

Każda dyrektywa rozpoczyna się znakiem # w pierwszej kolumnie. Po znaku # mogą wystąpić białe znaki.
Po dyrektywie może wystąpić znak komentarza // lub /* */
Dyrektywy mogą być specyfikowane w kilku liniach. Należy wówczas użyć znaku \ na końcu linii.

```
#define swapint(A,B)   \
{                      \
    int tmp=A;         \
    A=B;               \
    B=tmp;             \
}
```

Dyrektywy preprocesora mogą wystąpić w dowolnym miejscu kodu źródłowego. Odnoszą się wówczas do reszty pliku
Dyrektywy

```
#define                  #ifndef
#elif                    #import
#else                    #include
#endif                   #line
#error                   #pragma
#if                      #undef
#ifdef
```

Dyrektywa `#define`

Dyrektywa pozwala na zdefiniowanie nazwy (identyfikatora) dla stałej.
Składnia

`#define identifier token-string`$_{opt}$

lub

`#define identifier(identifier`$_{1,opt}$`,…, identifier`$_{n,opt}$`) token-string`$_{opt}$

Definicja pierwsza pozwala na zastąpienie każdego symbolu `identifier` przez ciąg symbolów `token-string`$_{opt}$.

Jeżeli ciąg symbolów `token-string`$_{opt}$ jest pusty, nazwa `identifier` pozostaje zdefiniowana, jednak jej każde wystąpienie jest usuwane z pliku źródłowego.

Przykład

```
#define PRINTABLE_AREA 100
#define PAGE_WIDTH PRINTABLE_AREA+20
```

Druga definicja specyfikuje tzw. makro. Każdemu wystąpieniu symbolu `identifier` musi towarzyszyć podanie w nawiasach parametrów `identifier`$_{1,opt}$`,…, identifier`$_{n,opt}$. Podczas wstawiania ciągu symbolów `token-string`$_{opt}$ do tekstu programu, nazwy użytych parametrów zastępują symbole `identifier`$_{1,opt}$`,…, identifier`$_{n,opt}$. Proces ten nazywa się *rozwijaniem makr.*

Przykład

```
int x=2,y=3;
swapint(x,y);
```

Jest zamieniane na

```
int x=2,y=3;
{int tmp=x; x=y; y=tmp;};
```

Uwagi

- Zazwyczaj kompilatory pozwalają na zdefiniowanie stałych preprocesora w linii komend. Mają one zastosowanie przy sterowaniu kompilacją warunkową. Stałe te można wprowadzać w odpowiednich oknach dialogowych IDE (*integrated development environment*).

- Kompilatory definiują także stałe identyfikujące typ kompilatora i wersję. Umożliwia to pisanie kodu adaptowanego do różnych kompilatorów:

```
#if defined BORALNDC
…
#endif
#if defined MSC_VER
…
#endif
```

Dyrektywy `#if, #elif, #else i #endif`

Powyższe dyrektywy sterują kompilacją warunkową. Pozwalają one na wyłączenie fragmentów kodu z procesu kompilacji.
Składnia

if-block

elif-blocks$_{opt}$

else-block$_{opt}$

endif-line

if-block:

`#if` *constant-expression text*

`#ifdef` *identifier*

`#ifndef` *identifier*

elif-blocks:

`#elif` *constant-expression text*

elif-blocks **`#elif`** *constant-expression text*

else-block:

`#else` *text*

endif-line:

`#endif`

Podsumowując:

- Każdej dyrektywie `#if` (`#ifdef` `#ifndef`) musi odpowiadać dyrektywa `#endif` domykająca blok tekstu kompilowany warunkowo.
- Pomiędzy parą (`#if, #endif`) może się znaleźć dowolna liczba bloków `#elif`.
- Bezpośrednio przed dyrektywą `#endif` może wystąpić dokładnie jeden blok `#else`.
- Tekst umieszczony w poszczególnych blokach może zawierać linie kodu źródłowego oraz kolejne dyrektywy preprocesora.
- Preprocesor na podstawie niezerowej wartości *constant-expression* w wyrażeniu warunkowym wybiera tekst do dalszego przetwarzania i przesyła go do kompilatora.

Przykład

```
#if LIBRARY_VERSION >= 4
// tekst przetwarzany dla wersji 4 i wyzej
#elif LIBRARY_VERSION == 3
// tekst przetwarzany dla wersji 3
#elif LIBRARY_VERSION == 2
// tekst przetwarzany dla wersji 2
#else
// tekst przetwarzany dla pozostalych wersji
#endif
```

Wyrażenia warunkowe

- Wyrażenia warunkowe *constant-expression* są wyrażeniami typu całkowitego.
- Mogą zawierać wyłącznie stałe całkowite, znakowe lub operator **`defined`** (testujący, czy dany identyfikator jest zdefiniowany)
- Wyrażenia mogą być skonstruowane z użyciem operatorów arytmetycznych lub logicznych.

Przykłady:

```
#define D 2
#if D>0
// ten blok jest przetwarzany przez kompilator
#endif

#if Z>0
// ten blok nie jest przetwarzany przez kompilator
// identyfikator Z nie jest zdefiniowany
#endif

#define X
#if defined X
// ten blok jest przetwarzany,
// ponieważ nazwa X jest zdefiniowa
#endif
#if !defined X
// ten blok nie jest przetwarzany
// ponieważ nazwa X jest zdefiniowana
#ednif
#if 0
// ten blok nigdy nie jest przetwarzany
#endif
```

Dyrektywy `#ifdef` **oraz** `#ifndef`

`#ifdef` *identifier* jest równoważne

`#if defined` *identifier*

`#ifndef` *identifier* jest równoważne

`#if !defined` *identifier*

Dyrektywa `#undef`

Składnia

`#undef` *identifier*

Dyrektywa usuwa wcześniej zdefiniowaną nazwę *identifier*. Jej podstawowym zastosowaniem jest możliwość zdefiniowania identyfikatora (stałej lub makra) o lokalnym zasięgu.

Przykład:

```
#define SIZE 10
char table[SIZE+1];
#undef SIZE



void f()
{
#define SIZE(A) sizeof(A)/sizeof(A[0])
    int i;
    for(i=0;i<SIZE(table);i++){
        table[i]=0;
    }
}
```

Dyrektywa `#error`

Dyrektywa powoduje wypisanie komunikatu o błędzie i przerywa proces kompilacji. Zazwyczaj używana jest do sprawdzenia, czy spełnione są określone warunki podczas kompilacji kodu (np.: zdefiniowane odpowiednie stałe).

```
#if defined BORALNDC
…
#elif defined MSC_VER
…
#else
#error This code must be compiled with\
Borland or Microsoft compiler
#endif
```

Dyrektywa `#include`

Składnia:

`#include` <*filename*>
`#include` "*filename*"

Obie postaci pozwalają na włączenie pliku *filename* w miejscu wystąpienia dyrektywy. W pliku *filename* mogą się znaleźć kolejne dyrektywy `#include` – czyli mogą być włączane kolejne pliki.

Specyfikacja *filename* może zawierać wyrażenie ścieżkowe. Jego konstrukcja powinna być zgodna z zasadami konstrukcji wyrażeń ścieżkowych dla plików w danym systemie operacyjnym.

Przykład:

```
#include <stdio.h>
#include "list.h"
#include "c:\myprojects\project1\list.h"
#include "library\list.h"
#include "/usr/john/projects/common/list.h"
#include "../include/list.h"
```

W jaki sposób preprocesor szuka plików.

Kompilator jest zazwyczaj odrębnym programem, który może być wywołany bezpośrednio z linii komend (np.: `bc`, `gcc`, `cl`, `cc`).

Jeżeli pracujemy w IDE (*integrated development environment*) wołany jest kompilator, ale jego wyjście jest przekierowywane do odpowiedniego okna.

Jedną z opcji kompilatora (zazwyczaj `-I` lub `/I` jest nazwa katalogu, w którym znajdują się standardowe pliki nagłówkowe (jak *stdlib.h*). W IDE nazwy te można podać w odpowiednim oknie dialogowym.

Dodatkowo w systemie może być zdefiniowana zmienna systemowa `INCLUDE` określająca listę katalogów, które zawierają pliki nagłówkowe.

- Jeżeli specyfikacja włączanego pliku ma postać
  `#include` <*filename*>
  wówczas plik szukany jest wyłącznie w katalogach standardowych określonych przez opcje `/I` kompilatora oraz zmienną systemową `INCLUDE.`
- Jeżeli specyfikacja włączanego pliku ma postać
  `#include "`*filename*`"`
  wówczas wpierw włączany plik *filename* jest szukany w kartotece, w której umieszczony jest kompilowany plik zawierający deyrektywę `#include`, a następnie dalej szukany jest w katalogach standardowych.
- Po znalezieniu pliku dalsze poszukiwania są przerywane.
- Jeżeli włączany plik nie został znaleziony, kompilacja jest przerywana i zwracany błąd.

Z tego powodu włączając pliki biblioteczne używamy zazwyczaj nawiasów ostrych, natomiast włączając własne pliki nagłówkowe używamy cudzysłowów:

```
#include <stdio.h>
#include "list.h"
```

Zabezpieczenie przed wielokrotnymi definicjami typów

Kompilator nie zezwala na wielokrotne definiowanie typów o tych samych nazwach. Zazwyczaj definiując nowy typ, umieszczamy jego definicję w pliku nagłówkowym i włączamy go w kilku modułach.

Budując aplikację złożoną z wielu modułów możemy być zmuszeni wielokrotnie włączyć ten sam nagłówek (bezpośrednio lub pośrednio).

Typowym zabezpieczeniem przed wielokrotnym przetwarzaniem tego samego nagłówka jest zdefiniowanie unikalnej stałej skonstruowanej na podstawie nazwy pliku nagłówkowego i testowanie, czy jest ona zdefiniowana.

Przykład:

list.h

```
#if !defined _list_h_
#define _list_h_
struct listelement
{
struct listelement*next;
int data;
};
struct list
{
struct listelement *head;
};
#endif
```

listmethods.h

```
#if !defined _listmethods_h_
#define _listmethods_h_
#include "list.h"
void init(struct list*);
void add(struct list*,int);
#endif //_listmethods_h_
```

main.c

```
#include "list.h"
#include "listmethods.h"
void main()
{
//…
}
```

Dyrektywa #pragma

Każdy z kompilatorów posiada szereg opcji sterujących przebiegiem kompilacji, które są ściśle związane ze środowiskiem działania, architekturą sprzętu i systemem operacyjnym.

Dyrektywa `#pragma` pozwala na uaktywnienie tych opcji za pomocą komend umieszczonych bezpośrednio w kodzie źródłowym kompilatora. Dzięki temu dla każdej jednostki kompilacji, a nawet fragmentów kodu źródłowego można ustawić indywidualne opcje. Równocześnie zachowana jest ogólna zgodność ze standardem języka C/C++.

Składnia

`#pragma` *ciąg-symbolów*

Zazwyczaj opcje kompilatorów nieco różnią się między sobą, stąd każdy kompilator definiuje i rozpoznaje inne pragmy.

Typowe pragmy (przykłady):

- *warning* – ustawia poziom szczegółowości raportowania błędów; za jej pomocą można wyłączać ostrzeżenia
- *pack* – ustawia sposób pakowania struktur
- *intrinsic* – zamiast generacji w kodzie wywołań niektórych prostych funkcji (jak `strlen`, `strcpy`, `strcmp`, `abs`) ich kod jest bezpośrednio wstawiany do kodu wynikowego. W wielu przypadkach pozwala to na przyspieszenie działania programu.
- *message* – wypisanie komunikatu

Szczegółową listę pragm rozpoznawanych przez kompilator i ich składnię należy sprawdzić w dokumentacji kompilatora. Dobrym zwyczajem jest umieszczanie dyrektyw w blokach kompilacji warunkowej.

```
#if defined _MSC_VER
#pragama ...// specyficzna do kampilatora Microsoft
#endif


#if defined _BORLANDC_
#pragama ...// specyficzna do kampilatora BORLAND
#endif
```

Makra

Makra definiuje się za pomocą dyrektywy `#define`. W odróżnieniu od definicji stałych muszą być użyte nawiasy. Wewnątrz nawiasów mogą pojawiać się nazwy formalnych parametrów makra (bez podania typów).

Jeżeli preprocesor rozpozna wystąpienie makra w kodzie źródłowym, jest ono zastępowane przez *ciało makra*. Jeżeli makro przyjmuje argumenty, wówczas formalne parametry są zastępowane argumentami wywołania. Proces ten nazywany jest *rozwijaniem makr*.

Preprocesor rozwija makra we wszystkich wierszach kodu źródłowego, które nie są dyrektywami i nie są wyłączone z kompilacji za pomocą dyrektyw kompilacji warunkowej.

Predefiniowane stałe

Kompilator definiuje zazwyczaj pewien zbiór stałych. W standardzie ANSI zdefiniowano sześć stałych. Pozostałe zależą od konkretnej implementacji kompilatora i mogą służyć do identyfikacji środowiska, w którym kod źródłowy jest kompilowany.

Podstawowy zestaw stałych

`__DATE__`	łańcuch znaków określający datę kompilacji danego pliku źródłowego w formacie *Mmm dd yyyyy*
`__FILE__`	nazwa pliku źródłowego
`__LINE__`	numer bieżącego wiersza kodu źródłowego
`__STDC__`	deklaruje pełną zgodność ze standardem ANSI
`__TIME__`	łańcuch znaków określający czas kompilacji pliku źródłowego w formacie *hh:mm:ss*
`__TIMESTAMP__`	data i czas ostatniej modyfikacji kodu źródłowego

Operatory

Preprocesor rozpoznaje trzy podstawowe operatory

defined – testowanie, czy identyfiaktor jest zdefiniowany

– konwersja tekstu na łańcuch znaków (*stringizing operator)*

– sklejanie symboli (*token-pasting operator*)

Operator #

Operator ten konwertuje parametr makra do postaci stałej tekstowej. Podczas rozwijania makra, jego parametr poprzedzony operatorem # jest zastępowany przez wartość tekstową argumentu użytego w wywołaniu. Utworzona stała łańcuchowa zawiera tekst definiujący argument makra ujęty w podwójne cudzysłowy.

Dodatkowo następuje konwersja znaków specjalnych definiowanych przez sekwencje *escape* (np.: '\\' , '\”' , '\'').

Przykład:

```
#define tostring( A ) #A
#define trace( A ) printf( "\n" #A "=%d\n",A )
void main()
{
    int x=3;
    char buf[256]="test";
    printf("%s\n",tostring(buf)); // wypisze ‘buf’
    printf("%s\n",buf);           // wypisze ‘test’
    trace(x);          // wypisze ‘x=3’
}
```

Operator ##

Operator ten pozwala na połączenie ze sobą wydzielonych symboli w jeden symbol.

1. Jeżeli przed lub po formalnym parametrem makra występuje operator ##, wówczas podczas rozwijania jest on zamieniany na formalny argument wywołania makra.
2. Operatory ## są następnie usuwane z tekstu, natomiast oddzielone nimi symbole sklejone w jeden.
3. Symbol powstały ze sklejenia jest dalej skanowany, aby stwierdzić, czy reprezentuje on zdefiniowane wcześniej makro. W takim przypadku makro jest również rozwijane.

Przykład 1

```
#define printvar(n) printf( "var" #n "= %d", var##n);
void main()
{
    int var3=3;
    printvar(3);
}
```

Przykład 2 (symulacja szablonów C++)

```
#define ListElementDef(A)                 \
typedef struct tagListElement##A{\
    struct tagListElement##A*next;    \
    A value;                          \
}ListElement##A

#define ListElement(A) ListElement##A

#define ListDef(A)                  \
typedef struct tagList##A{          \
    ListElement(A)*head;            \
}List##A
```

```
#define List(A) List##A

#define initListDef(A)              \
void initList##A(List(A)*list)    \
{                          \
    list->head=0;          \
}

#define initList(A) initList##A

/* definicja elementu listy, listy,
ciało funkcji initList() */
ListElementDef(int);
ListDef(int);
initListDef(int);

void main()
{
    List(int) list;
    initList(int)(&list);
    ListElement(int)*n=(ListElement(int)*)
        malloc(sizeof(ListElement(int)));
    n->value=1;
    list.head=n;
    printf("%d",list.head->value);
}
```

Program make

- Typowy program w języku C/C++ składa się z wielu odrębnych modułów (jednostek translacji). Ich liczba może dochodzić do kilkuset. Każdy z modułów jest kompilowany do postaci wynikowej OBJ; następnie wszystkie moduły są konsolidowane w jeden program wykonywalny (EXE) lub bibliotekę.
- Optymalizacja procesu kompilacji polega na tym, aby kompilować wyłącznie zmodyfikowane moduły. Podstawowym kryterium jest atrybut określający datę i czas ostatniej modyfikacji. Jeżeli plik OBJ nie istnieje lub jest on starszy niż plik źródłowy C/C++, wówczas powinien zostać on ponownie skompilowany.
- Następnie uaktualnione pliki OBJ są powtórnie konsolidowane.
- Nowoczesne kompilatory oferują dodatkową opcję: *inkremetalnego linkowania*. Proces konsolidacji polega na wymianie w programie wykonywalnym wyłącznie tych fragmentów kodu, który został zmodyfikowany. Nie jest przeprowadzana pełna konsolidacja.
- Zazwyczaj IDE (*integrated development environment*) ma wbudowane funkcje, które określają, które moduły należy skompilować. Przed powstaniem tego typu środowisk rolę tę pełnił program *make*. Bardzo często programy IDE generują plik konfiguracyjny (*makefile*) i następnie wywołują program *make* podczas kompilacji.

Działanie programu make

Program *make* uruchamia się podając nazwę pliku konfiguracyjnego. (Bardzo często pliki konfiguracyjne mają rozszerzenie *mak*). Jeżeli nazwa nie podana, standardowo czytany jest plik o nazwie *makefile* z bieżącego katalogu.

Przykład makefile

```
# komentarz
hello.exe : hello.c
    ccompiler.exe hello.c
```

Makefile specyfikuje *zależności* pomiędzy plikami. Tutaj istnieje zależność pomiędzy plikiem wykonywalnym `hello.exe` oraz plikiem źródłowym `hello.c`. Plik `hello.exe` jest nazywany obiektem docelowym (ang. *target*). Po dwukropku umieszcza się jeden lub więcej obiektów, od których obiekt docelowy jest zależny (ang. *dependency*). Jeżeli obiekt docelowy (`hello.exe`) nie istnieje lub jest starszy obiekty, od których jest uzależniony (plik `hello.c`), wykonywana jest *reguła* podana bezpośrednio po zależności. W podanym przykładzie regułą jest uruchomienie programu `ccompiler.exe` z argumentem `hello.c.`
Większość programów *make* wymaga, aby linia, w której opisane są reguły rozpoczynała się pojedynczym znakiem tabulacji.

Makra w programie make

W makefile można posługiwać się makrami. Makra nadają nazwy łańcuchom tekstowym. Aby rozwinąć makro, należy umieścić je wewnątrz nawiasów i poprzedzić znakiem `$`.
Przykład

```
# komentarz
CC = ccompiler.exe
hello.exe : hello.c
    $(CC) hello.c
```

Makra pozwalają na łatwiejszą modyfikację makefile. Na przykład nazwa i położenie kompilatora oraz jego opcje mogą być wprowadzone tylko raz.
W programie make istnieją także predefiniowane makra związane ze specyfikacjami plików występujących wyłącznie zależnościach.

`$@` Pełna ścieżka dla obiektu (pliku) docelowego.

`$*` Ścieżka i nazwa dla obiektu (pliku) docelowego bez rozszerzenia.

`$**` Wszystkie pliki, od których bieżący obiekt docelowy jest zależny.

`$?` Wszystkie pliki, od których bieżący obiekt docelowy jest zależny z późniejszym czasem modyfikacji (późniejszą pieczątką czasową, ang. *timestamp*).

`$<` Plik, od którego bieżący obiekt docelowy jest zależny z późniejszą pieczątką czasową.

Reguły dla rozszerzeń

Bardzo często wykonanie reguły polega na stworzeniu pliku o tej samej nazwie, ale zmienionym rozszerzeniu. Możemy skonstruować zależności, które będą działały dla plików o różnych nazwach, ale o znanych rozszerzeniach.

Przykład

```
CC = ccompiler.exe
OBJFLAG = -o # generate obj flag
INCLUDE = c:\ccompiler\include
LINK = linker.exe
.SUFFIXES: obj. c.
.c.obj:
    $(CC) -I$(INCLUDE) $(OBJFLAG) $*.c
test.exe : main.obj list.obj personlist.obj
    $(LINK) -o test.exe main.obj list.obj
personlist.obj
```

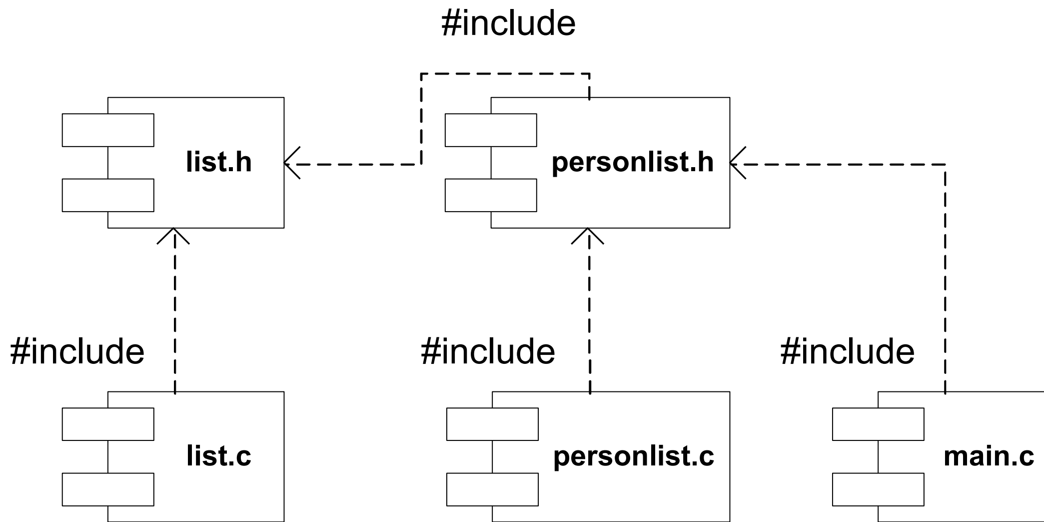
Powyższy przykład:

- Ustala zależność pomiędzy `test.exe` i modułami `main.obj list.obj personlist.obj`. Aby stworzyć plik wykonywalny należy wywołać program określony przez makro `LINK`.
- Aby stworzyć plik o rozszerzeniu *obj* z pliku p rozszerzeniu *c*, należy wywołać kompilator `$(CC)`, podając standardową kartotekę skąd należy czytać włączane pliki `$(INCLUDE)` z flagą `$(OBJFLAG)` generacji plików *obj*.

- Jeżeli którykolwiek z plików `main.c`, `list.c`, lub `personlist.c` będzie starszy niż odpowiadający im plik *obj*, wówczas zostanie on skompilowany. Plik *obj* będzie wówczas młodszy niż plik `test.exe`; wykonana zostanie reguła `$(LINK) -o test.exe main.obj list.obj personlist.obj`

Makefile i włączane pliki

Rozważmy przykład, w którym występują pliki nagłówkowe i pliki źródłowe.



- Plik `list.c` włącza za pomocą dyrektywy `#include` plik `list.h`.
- Plik `personlist.h` włącza `list.h`
- Plik `personlist.c` włącza `personlist.h`, a więc pośrednio także `list.h`.
- Plik `main.c` włącza `personlist.h`, a więc pośrednio także `list.h`.

Zastosowanie reguł opartych na rozszerzeniach plików nie zapewnia rekompilacji modułów w przypadku modyfikacji plików nagłówkowych. Na przykład po zmodyfikowaniu `personlist.h` należy powtórnie skompilować `personlist.c` oraz `main.c`, natomiast po modyfikacji `list.h` wszystkie moduły.

Poprawnie skonstruowany plik konfiguracyjny *makefile* powinien uwzględniać także zależności pomiędzy plikami zawierającymi kod źródłowy i plikami nagłówkowymi.

Przykład

```
CC = ccompiler.exe
OBJFLAG = -o # generate obj flag
INCLUDE = c:\ccompiler\include
LINK = linker.exe
list.obj : list.c list.h
    $(CC) -I$(INCLUDE) $(OBJFLAG) list.c
personlist.obj : personlist.c personlist.h list.h
    $(CC) -I$(INCLUDE) $(OBJFLAG) personlist.c
main.obj : main.c personlist.h list.h
    $(CC) -I$(INCLUDE) $(OBJFLAG) main.c
test.exe : main.obj list.obj personlist.obj
    $(LINK) -o test.exe main.obj list.obj
personlist.obj
```

Tego typu programy *makefile* są zazwyczaj generowane automatycznie przez IDE w procesie przeszukiwania drzewa zależności (ang. *scanning dependencies*). Podczas przeszukiwania włączane są kolejne pliki i analizowane zawarte w nich dyrektywy `#include`.

Sztuczny obiekt docelowy

Bardzo często dostarczana aplikacja składa się z pewnej liczby plików wykonywalnych lub bibliotek DLL. W jednym pliku konfiguracyjnym *makefile* możemy określić, jak zbudować je wszystkie. W tym celu specyfikujemy sztuczny obiekt docelowy (ang. *pseudotarget*) uzależniony od nich wszystkich. Odpowiadająca mu reguła jest pusta.

Przykład:

```
example1.exe : lista modułów
    reguła tworząca example1.exe
example2.exe : lista modułów
    reguła tworząca example2.exe
example3.exe : lista modułów
    reguła tworząca example3.exe
all: example1.exe example2.exe example3.exe
```

Aby stworzyć sztuczny obiekt docelowy `all`, wykonywane będą reguły tworzące `example1.exe`, `example2.exe` oraz `example3.exe.` Plik `all` nigdy nie powstanie, ponieważ reguła jest pusta.

Przy tak skonstruowanym pliku *makefile* możliwe jest dalej generowanie indywidualnych plików wchodzących w skład aplikacji (lub plików wynikowych *obj*). Jednym z parametrów komendy *make* jest nazwa obiektu docelowego.

- w wyniku wywołania `make example2.exe` po przeczytaniu pliku *makefile* wykonane zostaną reguły budujące wyłącznie program wykonywalny `example2.exe;`
- w wyniku wywołania `make` lub `make all` wykonane zostaną reguły budujące wszystkie pliki określone przez zależność: `all: example1.exe example2.exe example3.exe`

Podsumowanie

Program *make* lub wzorowane na nim rozwiązanie stsowane w środowiskach IDE, jest narzędziem umożliwiającym optymalizację procesu kompilacji dużych programów, składających się z większej liczby modułów.

Kryterium decydującym o tym, czy dany moduł powinien zostać skompilowany jest czas modyfikacji (pieczątka czasowa) pliku źródłowego lub włączanych do niego plików nagłówkowych.

Poprawne działanie programu *make* (a także IDE) jest uzależnione od poprawnych informacji o aktualnym czasie stacji roboczej i czasie modyfikacji plików źródłowych.

Program *make* może mieć bardziej ogólne zastosowanie wykraczające poza proces kompilacji programów w języku C/C++. Na przykład sporządzanie kopii zapasowych, archiwizacja plików, zmiana formatu plików, itd.

Często używane funkcje

Funkcje konwersji

```
int atoi(const char*string)
double atof(const char*string)
long atol(const char*string)
```

Każda z funkcji konwertuje łańcuch znaków na liczbę typu `int`, `double` lub `long`. Funkcje zwracają 0 w przypadku, jeżeli podany tekst nie może zostać przekonwertowany, co nie pozwala na odróżnienie kodu błędu od rezultatu konwersji tekstu `"0"`. Zdefiniowane w `<stdlib.h>`. Występują w standardzie ANSI.

```
char *itoa(int value, char*string, int radix);
```

Funkcja konwertuje liczbę całkowitą `value` do postaci łańcucha znaków. Parametr `string` specyfikuje bufor, gdzie ma być składowany rezultat. Parametr `radix` (z zakresu 2 – 36) określa podstawę systemu konwersji.

```
char *gcvt(double value, int digits, char *buffer);
```

Konwertuje liczbę typu `double` do postaci tekstowej. Parametr `digits` określa liczbę znaczących cyfr. Przekonwertowana liczba jest składowana w tablicy znaków `buffer`.

Zdefiniowane w `<stdlib.h>`.

Funkcje systemowe

getenv

```
char * getenv(const char*varname);
```

Funkcja odczytuje wartość zmiennej systemowej. Systemy operacyjne oferują możliwość zdefiniowania zmiennych systemowych, czyli przypisania nazwom tekstów (zazwyczaj są to nazwy katalogów). Definiują one środowisko wykonania programów.
Funkcja zwraca wartość zmiennej, jeżeli zmienna `varname` jest zdefiniowana. W przeciwnym wypadku zwraca wartość 0 (`NULL`).

putenv

`int putenv(const char*envstring);`

Funkcja pozwala zmodyfikować (stworzyć, usunąć) zmienne systemowe.

Tablica `envstring` powinna mieć postać :
`"nazwa_zmiennej=tekst"`,
na przykład:
`“PATH=C:\WINNT\system32;C:\WINNT;”`
Postać powinna być zgodna ze składnią interpretera poleceń systemu operacyjnego.

Przykład

```
void main(){
    char *path;
    path = getenv( "PATH" );
    if( path != NULL ){
        char envstring[4096];
        sprintf(envstring,
            "PATH=%s;%s",path,"c:\\mypath");
        putenv( envstring );
        path = getenv( "PATH" );
        printf( "PATH=%s\n", path );
    }
}
```

Uwaga: funkcja nie zmienia globalnych wartości zmiennych systemowych, ale wartości, które są widziane przez dany proces i jego procesy potomne.

system

`int system(const char*command);`

Funkcja wykonuje polecenie systemu operacyjnego (w tym dowolny program wykonywalny). Funkcja zwraca 0, jeżeli interpreter poleceń zwrócił wartość 0, wartość –1 oznacza błąd.

Przykład:

```
system ("dir c:\\*.* | more" );
system ("mojprogram.exe < in.txt > out.txt" );
```

Uwagi
Dla platformy WIN32 istnieją dwie grupy funkcji `spawn` i `exec`, które pozwalają uruchamiać nowe procesy, przekazywać w różny sposób do nich argumenty i szczegółowo kontrolować synchronizację działania procesu potomego i procesu wołającego (rodzica). Nie należą one do standardu ANSI.

Uruchamianie wątków

Platforma WIN32 definiuje dwie funkcje od obsługi wątków: `_beginthread` i `_endthread`.

Funkcja **`_beginthread`**

```
unsigned long _beginthread(
void( __cdecl *thread_function )( void * ),
unsigned stack_size,
void *arglist );
```

`thread_function` – adres funkcji, która ma być uruchomiona jako wątek. Jest ona zadeklarowana jako:
`void thread_function(void *arglist);`
`stack_size` – rozmiar stosu (0 – należy użyć wartości domyslnej)
`arglist` – lista argumentów dla funkcji wątky

Funkcja **`_endthread`**
`void _endthread(void);`
Funkcja kończy bieżący wątek.

Poniższy przykład używa funkcji WIN32 `Sleep`, która zawiesza wykonanie wątku na określony czas. Parametr jest podany w milisekundach. Wywołanie `Sleep(0)` jest równoznaczne natychmiastowemu przekazaniu sterowania do innego wątku.

Przykład:

```
#include <windows.h>
#include <winbase.h>
#define PHIL_COUNT 5
int forks[PHIL_COUNT];

void philospoher(void*vnumber)
{
    int i;
    int number=(int)vnumber;
    int leftFork=number;
    int rightFork=(number+1)%PHIL_COUNT;
    for(;;){
        printf("Philospoher %d thinks\n",number);
        Sleep(300);
        // wait for left fork
        while(forks[leftFork]==1){/*Sleep(0)*/}
        forks[leftFork]=1;
        // wait for right fork
        while(forks[rightFork]){ /*Sleep(0)*/}
        forks[rightFork]=1;
        printf("Philospoher %d eats\n",number);
        Sleep(300);
        forks[rightFork]=0;
        forks[leftFork]=0;
    }
}

#include <process.h>
void main()
{
    int i;
    for(i=0;i<PHIL_COUNT;i++){
        _beginthread(philospoher,0,(void*)i);
    }
    // petla nieskończona w wątku głównym
    for(;;);

}
```

Funkcje działające na nazwach plików

Funkcja _fullpath

Funkcja pozwala na ustalenie pełnej (tzw. absolutnej ścieżki) pliku na podstawie ścieżki relatywnej. (Platforma WIN32, nagłówek <stdlib.h>)

```
char *_fullpath( char *absPath, const char *relPath,
size_t maxLength );
```

absPath – wskaźnik do bufora, w którym zostanie umieszczona absolutna ścieżka

maxLength – długość bufora

relPath – łańcuch znaków określający relatywną ścieżkę.

Funkcja zwraca 0 (NULL), w przypadku błędów – np.: plik nie może zostać odnaleziony lub długość bufora jest zbyt mała.

Przykład:

```
#include <stdlib.h>
void main()
{
    char full [_MAX_PATH];
    fullpath(full,”..\\example.txt”,MAX_PATH);
    printf(full);
}
```

Funkcja _splitpath

Funkcja dzieli ścieżkową nazwę pliku na składowe (nazwa dysku, ścieżka, nazwa pliku, rozszerzenie)

```
void _splitpath( const char *path,
char *drive, char *dir, char *fname, char *ext );
```

`path` – nazwa pliku
`drive` – bufor, w którym będzie umieszczona nazwa napędu (dysku). Powinien mieć rozmiar `_MAX_DRIVE`.
`dir` – bufor, w którym będzie umieszczona ścieżka pliku (bez napędu). Powinien mieć rozmiar `_MAX_DIR`.
`fname` – bufor, w którym będzie umieszczona nazwa pliku. Powinien mieć rozmiar `_MAX_FNAME`.
`ext` – bufor, w którym będzie umieszczone rozszerzenie. Powinien mieć rozmiar `_MAX_EXT`.

Funkcja _makepath

Funkcja wykonuje operację odwrotną: składa nazwę pliku z komponentów.

```
void _makepath( char *path,
const char *drive, const char *dir,
const char *fname, const char *ext );
```

`path` – bufor, w którym będzie umieszczona ścieżkowa nazwa pliku. Powinien mieć rozmiar `_MAX_PATH`.
`drive` – wskaźnik na tekst określający literę napędu (dwukropek może zostać pominięty)
`dir` – wskaźnik na tekst określający ścieżkę. Znaki \ na początku i końcu nie są wymagane.
`fname` – wskaźnik na tekst określający nazwę pliku
`ext` – wskaźnik na tekst określający rozszerzenie. Początkowa kropka nie jest wymagana.
Jeżeli którykolwiek z komponentów jest wskaźnikiem zerowym (`NULL`) lub pustym tekstem, zostanie on pominięty przy konstrukcji ścieżki.

Przykład

```
void main(int argc, char**argv)
{
    char path[_MAX_PATH];
    char drive[_MAX_DRIVE];
    char dir[_MAX_DIR];
    char fname[_MAX_FNAME];
    char ext[_MAX_EXT];
    _splitpath( argv[0], drive, dir, fname, ext );
    _makepath( path, drive, dir, fname, "cfg" );
    printf("Config path: %s\n",path);

}
```

Wyliczenie zawartości katalogów

Wyliczenie plików i podkatalogów w danego katalogu umożliwiają funkcje `_findfirst`, `_findnext`, `_findclose` zdefiniowane w `<io.h>`.

1. Proces przeszukiwania katalogu rozpoczyna wywołanie funkcji `_findfirst` ze specyfikacją poszukiwanych plików jako argumentem (np.: `"*.doc")`. Specyfikacja może zawierać symbole wieloznaczne (? i *).
2. Następnie kolejne pliki są szukane za pomocą funkcji `_findnext`, dopóki nie zwróci ona wartości różnej od 0 wskazującej koniec przeszukiwania.
3. Na zakończenie proces przeszukiwania powinien zostać zamknięty przez wywołanie funkcji `_findclose`.

Funkcje `_findfirst` i `_findnext` wypełniają informacjami o znalezionych plikach strukturę `_finddata_t`. Zawiera ona:

- nazwę pliku (bez ścieżki, ale z rozszerzeniem);
- flagę określają atrybuty pliku (jest ona kombinacją stałych `_A_ARCH`, `_A_HIDDEN`, `_A_NORMAL`, `_A_RDONLY`, `_A_SUBDIR`, `_A_SYSTEM`)
- informacje o czasie utworzenia, ostatniego dostępu oraz zapisu
- rozmiar pliku w bajtach

Funkcja _findfirst

```
long _findfirst( char *filespec,
    struct _finddata_t *fileinfo );
```

`filespec` – specyfikacja szukanych plików

`fileinfo` – wskaźnik do struktury `_finddata_t`

Jeżeli istnieją pliki pasujące do specyfikacji, funkcja zwraca systemowy identyfikator (ang. *handle*) określający grupę szukanych plików. Identyfikator powinien być użyty w kolejnych wywołaniach `_findnext` oraz w `_findclose`. Równocześnie struktura `fileinfo` jest wypełniana o pierwszym pliku pasującym do specyfikacji.

Jeżeli takich plików brak funkcja zwraca wartość –1.

Funkcja _findnext

```
int _findnext( long handle,
    struct _finddata_t *fileinfo );
```

`handle` – identfikator systemowy dla procesu poszukiwania zwrócony przez funkcję `_findfirst`.
`fileinfo` – wskaźnik do struktury `_finddata_t`

Funkcja zwraca wartość 0, jeżeli znaleziono kolejny plik pasujący do specyfikacji określonej w wywołaniu `_findfirst`. W przeciwnym wypadku zwraca –1.

Funkcja **_findclose**

```
int _findclose( long handle );
```

`handle` – identfikator systemowy dla procesu poszukiwania zwrócony przez funkcję `_findfirst`.
Funkcja kończy proces przeszukiwania, zwalniając systemowe zasoby określone przez parametr `handle`.

Przykład

```
#include <io.h>
void main()
{
    struct _finddata_t fileinfo;
    long handle;
    handle = _findfirst( "c:\\*.*", &fileinfo );
    if(handle<0)return;
    printf((fileinfo.attrib&_A_SUBDIR?"[%s]\n":"%s\n"),
        fileinfo.name);
    while(_findnext(handle,&fileinfo)==0){
        printf((fileinfo.attrib&_A_SUBDIR?"[%s]\n":"%s\n"),
            fileinfo.name);
    }
    _findclose(handle);
}
```

Funkcja wypisze wszystkie pliki znajdujące się w katalogu głownym dysku c:. Nazwy podkatalogów zostaną ujęte w nawiasy. Zapewnia to wybór formatu dla funkcji `printf`:

```
(fileinfo.attrib&_A_SUBDIR?"[%s]\n":"%s\n")
```