Podstawy Programowania 1

Biblioteki

Arkadiusz Chrobot

Katedra Systemów Informatycznych

15 stycznia 2023

Plan

Biblioteki

Kod źródłowy dużych programów komputerowych jest zazwyczaj dzielony na osobne pliki, które ogólnie nazywane są jednostkami translacji, a w przypadku języków kompilowanych - jednostkami kompilacji. Celem takiego podziału jest nie tylko zwiększenie czytelności kodu poprzez pogrupowanie logicznie powiązanych elementów programów, ale również umożliwienie ich wielokrotnego wykorzystania w tworzeniu innego oprogramowania. Jednostki translacji zawierające funkcje, zmienne, typy danych i stałe do ponownego użycia nazywane są najczęściej bibliotekami. Nie są to jednak wyłącznie „pojemniki na kod". Pozwalają one bowiem sprawnie tym kodem zarządzać, pozwalając zadecydować programiście co chce udostępnić na zewnątrz biblioteki, a co zachować w ich wnętrzu.

Interfejsy i implementacje bibliotek

Elementy zgromadzone w bibliotece, które są udostępniane na zewnątrz niej, do publicznego użytku, nazywane są jej *interfejsem*, a elementy zamknięte w jej wnętrzu nazywamy *implementacją*. Aby taki podział był możliwy język programowania musi udostępniać programiście odpowiednie *mechanizmy ukrywania kodu*. Żeby wyjaśnić zasadność oddzielenia interfejsu od implementacji trzeba spojrzeć na problem budowy bibliotek z dwóch perspektyw: programisty-twórcy i programisty-użytkownika biblioteki. Dosyć często są to dwie różne osoby. Programista-użytkownik chce otrzymać bibliotekę, która będzie prosta i niekłopotliwa w użyciu. Chce on mieć klarowną informację, których elementów może użyć w swoim programie i w jaki sposób. Jeśli pojawi się nowsza wersja tej biblioteki z rozszerzonym interfejsem lub zmienioną implementacją, to program programisty-użytkownika powinien się dać z nią skompilować, bez konieczności wprowadzania poprawek do jego kodu źródłowego. Programista-twórca chce zachować możliwość wprowadzania poprawek w nowszych wersjach biblioteki, ale nie chce też utrudniać pracy programisty-użytkownika zmuszając go do modyfikacji jego programu.

Interfejsy i implementacje bibliotek

Kontynuacja

Aby spełnić wymagania programisty-użytkownika i swoje autor biblioteki powinien w jej interfejsie zawrzeć te elementy, które nie ulegną zmianie w jej kolejnych wersjach. Jeśli jednym z takich elementów jest funkcja, to sposób jej wywołania musi być ten sam w nowych wydaniach biblioteki, co oznacza, że jej nazwa, typ i kolejność parametrów muszą być zachowane. Interfejs jest odpowiedzialny za współpracę biblioteki z innymi jednostkami translacji, a więc może ulec w jej kolejnych wydaniach jedynie rozszerzeniu, nie może być zawężany, ani zmieniany. W implementacji biblioteki jej twórca powinien zawrzeć wszystkie te elementy, które stanowią mechanizm jej działania i które mogą ulec dowolnej zmianie w jej kolejnych wersjach. Przypomina to działania np. producenta samochodów, który w kolejnych modelach auta zachowuje układ kierowniczy, którym posługuje się kierowca, a może zmieniać inne elementy, jak np. układ napędowy.

Biblioteki w języku C

Język C pozwala na użycie w programach zarówno bibliotek statycznych, jak i dynamicznych. Pierwsze dołączane są do programu w trakcie jego kompilacji. Drugie są ładowane do pamięci komputera dopiero wtedy, gdy program odwoła się do jednego z elementów w nich zawartych. Ponadto mogą one być współużytkowane przez wiele programów jednocześnie. Ich proces tworzenia jest taki sam, ale różnią się sposobem użycia, który w przypadku bibliotek dynamicznych jest bardziej skompilowany i zależny od systemu operacyjnego. Jest to powód, dla którego na tym wykładzie przedstawione będą wyłącznie biblioteki statyczne.

Deklaracje i definicje zmiennych

Do tej pory miejsce określenia w programie zmiennej nazywaliśmy w uproszczeniu deklaracją. Jeśli zmienna jest używana wyłącznie w obrębie jednej jednostki translacji, to takie miejsce jest nie tylko jej deklaracją, ale również definicją. W przypadku, kiedy zmienna jest używana w wielu osobnych plikach miejsca jej definicji i deklaracji mogą być rozdzielne. Co więcej, zmienna może mieć wiele deklaracji, ale tylko jedno miejsce definicji. Deklaracja jest informacją dla kompilatora, że zmienna została zdefiniowana w innym pliku, ale w tym będzie używana. Definicja z kolei informuje kompilator, że musi w miejscu jej wystąpienia przydzielić pamięć na zmienną i nadać jej wartość początkową. Deklaracja różni się od definicji tym, że rozpoczyna się do słowa kluczowego `extern`. Zadeklarowana w ten sposób zmienna musi być zdefiniowana w jednej z jednostek translacji na jakie jest podzielony kod, inaczej program się nie skompiluje. Następny slajd zawiera przykład, w którym zdefiniowano i zadeklarowano zmienną w dwóch osobnych plikach.

Deklaracje i definicje zmiennych

Przykład

Plik 1: external.c

```
int foo;
```

Plik 2: program.c

```
#include<stdio.h>

extern int foo;

int main(void)
{
        printf("%d\n",foo);
        return 0;
}
```

Deklaracje i definicje zmiennych

Komentarz

Plik `external.c` zawiera definicję zmiennej o nazwie `foo`, która jest używana w pliku `program.c`. Ponieważ jest to zmienna globalna, to jej wartość początkowa będzie wynosiła zero.

Zmienne statyczne

Jeśli chcielibyśmy w pliku zdefiniować zmienną *globalną*, która nie byłaby dostępna na zewnątrz, to jej definicję poprzedzilibyśmy słowem kluczowym `static`. Dotychczas używaliśmy tego słowa wyłącznie w odniesieniu do zmiennych lokalnych. Przypomnijmy, że zmienna lokalna poprzedzona słowem `static` nie ulega zniszczeniu między kolejnymi wywołaniami funkcji, a jej wartość początkowa, tj. przed pierwszym wywołaniem funkcji, w której jest zadeklarowana i zdefiniowana, wynosi zero. W przypadku zmiennych globalnych to słowo ogranicza zakres ich widoczności (ang. *scope*) do pliku w którym są one zdefiniowane. Takie zmienne statyczne mogą być bezpośrednio używane w funkcjach zdefiniowanych wewnątrz pliku, gdyż oprócz tych funkcji żaden inny kod nie ma wpływu na ich zawartość.

Definicje i deklaracje funkcji

Podobnie jak zmienne, również funkcje mogą być definiowane i deklarowane w osobnych plikach. Sposób deklaracji funkcji był opisany w ramach wykładu poświęconego tym podprogramom. Przypomnijmy, że deklaracja funkcji składa się z jej nagłówka zakończonego średnikiem. Dodatkowo w deklaracji wolno nam pominąć nazwy parametrów funkcji, pozostawiając tylko ich typy. Deklarację funkcji, jeśli jej definicja znajduje się w osobnym pliku, można także poprzedzić słowem kluczowym `extern`, ale nie jest ono obowiązkowe. Jeśli chcemy uczynić funkcję dostępną jedynie w obrębie pliku, w którym jest zdefiniowana, to w jej nagłówku, przed definicją typu wartości przez nią zwracanej użyjemy słowa kluczowego `static`. Na podstawie tej i wcześniejszych informacji o słowie kluczowym `static` możemy stwierdzić, że służy ono do *ukrywania implementacji*, czyli szczegółów jednostek translacji, które nie powinny być dostępne na zewnątrz. Kolejny slajd zawiera przykład definicji i deklaracji pojedynczej funkcji umieszczonych w osobnych plikach.

Deklaracje i definicje funkcji

Przykład

Plik 3: external_2.c

```c
#include<stdio.h>

void print(int variable)
{
        printf("%d\n",variable);
}
```

Plik 4: program_2.c

```c
extern int foo;
extern void print(int);

int main(void)
{
        print(foo);
        return 0;
}
```

Deklaracje i definicje funkcji

Komentarz

Funkcja `print()` została zdefiniowana w pliku `external_2.c`. Ponieważ wywołuje ona funkcję `printf()`, to konieczne było włączenie pliku nagłówkowego `stdio.h`. Wywołanie funkcji `print()` następuje w pliku `program_2.c`, gdzie jest ona również deklarowana. Kod zawarty w tym pliku korzysta także ze zmiennej `foo`, której definicja znajduje się w pliku `external.c`.

Pliki nagłówkowe

Korzystanie ze zmiennych i funkcji zdefiniowanych w innych plikach wymaga powtórzenia ich deklaracji w każdej jednostce translacji, w której będą one użyte. Tę uciążliwość można zniwelować umieszczając te deklaracje w *pliku nagłówkowym*. Jest to plik z kodem źródłowym, który tradycyjnie ma nazwę zakończoną rozszerzeniem `.h`. Może on zawierać wiele deklaracji funkcji i zmiennych zdefiniowanych w co najmniej jednym pliku z rozszerzeniem `.c`, ale nie powinien zawierać ich definicji, ani żadnych innych instrukcji, które wiązałby się z przydziałem pamięci wewnątrz takiego pliku. Plik nagłówkowy może także zawierać inne elementy kodu źródłowego programu, takie jak definicje typów wyliczeniowych, struktur, unii, a także makra preprocesora, które będą opisane w dalszej części wykładu. Następny slajd zawiera przykład takiego pliku.

Pliki nagłówkowe

Plik 5: header.h

```
#ifndef HEADER_H
#define HEADER_H

extern int foo;
extern void print(int);

#endif
```

Plik nagłówkowe

Komentarz

W przedstawionym na poprzednim slajdzie przykładzie pliku nagłówkowego występują deklaracje funkcji i zmiennej znanych z poprzednich przykładów. Są one umieszczone wewnątrz dyrektyw (instrukcji) preprocesora, których celem jest zapewnienie, że deklaracje z pliku nagłówkowego zostaną włączone do programu tylko raz, mimo, że instrukcje jego włączenia mogą się pojawiać w programie wielokrotnie. Instrukcję `#ifndef` możemy odczytać jako „jeśli nie istnieje definicja". Występująca za nią nazwa, to znacznik, który nazywamy *wartownikiem pliku nagłówkowego* może on być dowolny, ale zazwyczaj tworzy się go tak, jak w podanym przykładzie. Instrukcja `#define` jest już nam znana, tym razem jednak nie jest użyta do definicji stałej, a wartownika pliku nagłówkowego, czyli pustej nazwy pełniącej rolę znacznika. Instrukcja `#endif` oznacza koniec fragmentu kodu objętego instrukcją `#ifndef`. Całościowo zapis ten należy rozumieć następująco: jeśli nie jest zdefiniowany wartownik, to go zdefiniuj i włącz do programu wszystko to, co znajduje się przed dyrektywą `#endif`, w przeciwnym przypadku pomiń to wszystko i nie włączaj do kodu źródłowego programu.

Pliki nagłówkowe

Włączenie pliku nagłówkowego do programu

Włącznie pliku nagłówkowego do pliku z rozszerzeniem `.c` wykonywane jest przez preprocesor, czyli program uruchamiany w procesie kompilacji przed kompilatorem. Jeśli plik nagłówkowy nie zawiera makr, to dokonuje on jedynie skopiowania jego zawartości do wspomnianego pliku z rozszerzeniem `.c`, chyba, że plik nagłówkowy był już wcześniej kopiowany do programu. Instrukcją nakazującą preprocesorowi włączenie danego pliku nagłówkowego jest dyrektywa `#include`. Jeśli plik ten znajduje się w katalogu, który kompilator domyślnie przyjmuje jako miejsce składowania takich plików, to nazwa pliku nagłówkowego jest ujęta w nawiasy ostrokątne. W przeciwnym przypadku powinna być umieszczona w cudzysłowach. W niektórych sytuacjach konieczne jest podanie w tej dyrektywie pełnej ścieżki dostępu do pliku. Plik nagłówkowy może być włączony również do pliku, gdzie znajdują się definicje zadeklarowanych w nim zmiennych i funkcji. Umożliwi to kompilatorowi sprawdzenie ich poprawności. W przypadku, gdy w tym pliku używane są definicje typów lub makr z pliku nagłówkowego, to włączenie tego ostatniego staje się koniecznością.

Pliki nagłówkowe

Włączenie plików nagłówkowych - uwaga

Z zamieszczonego na poprzednim slajdzie opisu działania mechanizmu włączania plików nagłówkowych można wyciągnąć bardzo ważny wniosek: Nie należy włączać plików nagłówkowych do programu, jeśli żaden element w nich zawarty nie będzie używany. Zwiększają one jedynie niepotrzebnie objętość takich programów. Następne slajdy zawierają przykład użycia pliku nagłówkowego wraz z odpowiednimi plikami z rozszerzeniem `.c`

Pliki nagłówkowe

Przykład

Plik 6: external2.c

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

void print(int variable)
{
        printf("%d\n",variable);
}
```

Pliki nagłówkowe

Poprzedni slajd zawiera treść zmodyfikowanego plik `external_2.c`. Została w nim dodana instrukcja włączająca plik nagłówkowy `header.h`, który w tym wypadku znajduje się w tym samym katalogu. To włączenie nie jest obowiązkowe, ale dzięki niemu kompilator będzie mógł sprawdzić, czy definicja i deklaracja funkcji się pokrywają.

Pliki nagłówkowe

Przykład

Plik 7: external.c

```c
#include"header.h"

int foo;
```

Pliki nagłówkowe

Komentarz

Poprzedni slajd zawiera kod źródłowy ze zmodyfikowanego pliku `external.c`. Podobnie jak w przypadku poprzednio opisywanego pliku włączenie pliku nagłówkowego nie jest konieczne, ale pozwala kompilatorowi sprawdzić poprawność deklaracji i definicji zmiennej.

Pliki nagłówkowe

Przykład

Plik 8: program2.c

```c
#include"header.h"

int main(void)
{
        print(foo);
        return 0;
}
```

Pliki nagłówkowe

Przykład

Pliki nagłówkowe wraz z odpowiadającymi im plikami z rozszerzeniem `.c` tworzą biblioteki. Program z poprzedniego slajdu demonstruje sposób użycia biblioteki statycznej. Najpierw włączany jest plik nagłówkowy z deklaracjami zmiennej `foo` oraz funkcji `print()`, a następnie te elementy programu są używane tak, jakby były zdefiniowane w pliku zawierającym funkcję `main()` programu.

Biblioteki - podsumowanie

Biblioteki w języku C składają się z pliku nagłowkowego, który może zawierać definicje typów i stałych oraz deklaracje zmiennych i funkcji oraz z plików, w postaci źródłowej lub skompilowanej, które zawierają definicje tych zmiennych i podprogramów.

Makra preprocesora

Obsługa plików nagłówkowych jest tylko jedynym z zadań preprocesora. Preprocesor ma swój własny język programowania, który nie jest jednak tłumaczony na kod maszynowy, a służy do sterowania procesem kompilacji programów. Częściowo już go poznaliśmy. Oprócz dyrektywy `#include` często korzystaliśmy z instrukcji `#define` do tworzenia stałych. Tak zdefiniowana stała nazywana jest też *makrem* lub *makrodefinicją*. Wszędzie gdzie wystąpi nazwa stałej w programie preprocesor zastąpi ją wartością, która została z nią związana. Makra nie muszą być jedynie prostymi definicjami stałych, mogą być rozbudowanymi podprogramami z własną listą parametrów. Nie są one wywoływane tak jak funkcje, lecz ich kod w całości jest umieszczany przez preprocesor w miejscu programu, gdzie występują ich nazwy z argumentami. Zaletą makr jest to, że nie wymagają przydzielenia miejsca na stosie, więc działają trochę szybciej niż funkcje. Z drugiej strony preprocesor nie dokonuje takiej kontroli typów wartości, jaką wykonuje kompilator, więc o ewentualnych błędach w makrach dowiadujemy się dopiero w trakcie kompilacji, a ich lokalizacja jest utrudniona z uwagi na zmiany dokonywane w kodzie przed kompilacją przez preprocesor.

Makra preprocesora

Definicje makr są najczęściej umieszczane w plikach nagłówkowych. Można je także definiować w plikach z rozszerzeniem .c. Dla uproszczenia prezentacji wszystkie makra, które zostaną pokazane na tym wykładzie będą właśnie definiowane w ten sposób. „Wywołanie" makra nazywa się rozwinięciem. Z uwagi na to, w jaki sposób ta czynność wykonywana jest przez preprocesor, należy zachować ostrożność podczas definiowania i stosowania makr. Kolejne slajdy zawierają kod źródłowy programu zawierającego definicję prostego makra, które może dawać dosyć nieoczekiwane wyniki działania. Zostanie przedstawiona również poprawiona wersja tego makra, która zachowuje się zgodnie z naszymi oczekiwaniami.

Makra preprocesora

Przykład

```c
#include<stdio.h>

#define MULTIPLY(X,Y) X*Y

int main(void)
{
    int a=2, b=3;
    printf("Wynik mnożenia %d\n",MULTIPLY(a,b));
    return 0;
}
```

Makra preprocesora

Komentarz

W programie zostało zdefiniowane makro o nazwie `MULTIPLY`, które mnoży przez siebie argumenty, które zostały przekazane mu przez parametry `X` i `Y`. Proszę zwrócić uwagę, że po wyrażeniu zawartym w makrze nie stawiamy średnika. Musimy o nim pamiętać w miejscu użycia tego makra. Warto także zauważyć, że nazwy makr i ich parametrów na mocy konwencji piszemy wielkimi literami, tak jak nazwy stałych. W zademonstrowanym przykładzie za parametry makra podstawiane są zmienne `a` i `b`, o wartościach odpowiednio 2 i 3. Zgodnie z oczekiwaniem program wypisze na ekranie wartość 6. Wystarczy jednak zmienić trochę kod tego programu, aby wynik działania makra zaczął odbiegać od naszych oczekiwań, co pokazano na następnym slajdzie.

Makra preprocesora

Przykład

```c
#include<stdio.h>

#define MULTIPLY(X,Y) X*Y

int main(void)
{
    int a=2, b=3;
    printf("Wynik mnożenia %d\n",MULTIPLY(a+1,b-1));
    return 0;
}
```

Makra preprocesora

Komentarz

Teraz program wypisze na ekranie 4 zamiast 6. Dzieje się tak z uwagi na to, że wartości argumentów makra, w przeciwieństwie do wartości argumentów funkcji, nie są wyliczane przed jego rozwinięciem. Zatem zamiast wyrażenia 3*2 preprocesor utworzy wyrażenie `a+1*b-1` „wklejając" wartości argumentów, z jakimi makro zostało użyte, w miejsce parametrów `x` i `y`. Ostatecznie, podczas wykonania programu otrzymamy wyrażenie 2+1*3-1, które faktycznie ma wartość 4. Aby uniknąć takich sytuacji wystarczy oba parametry makra otoczyć w wyrażeniu nawiasami okrągłymi, tak jak pokazano to na następnym slajdzie. Niestety, nie chroni to przez sytuacją, gdy za te parametry programista podstawi np. łańcuchy znaków. Preprocesor nie wykryje takiego błędu, zrobi to dopiero kompilator.

Makra preprocesora

Przykład

```c
#include<stdio.h>

#define MULTIPLY(X,Y) (X)*(Y)

int main(void)
{
    int a=2, b=3;
    printf("Wynik mnożenia %d\n",MULTIPLY(a+1,b-1));
    return 0;
}
```

Makra preprocesora

Makra preprocesora mogą zawierać więcej niż jedną instrukcję. Dodatkowo, każda z instrukcji zawartych w makrze może być zapisana w osobnym wierszu programu. Zapis definicji takiego makra jest jednak zupełnie inny niż funkcji. Jeśli makro ma więcej niż dwa wiersze kodu, to wszystkie wiersze między pierwszym, a ostatnim muszą się kończyć znakiem `\`, aby zaznaczyć, że kolejny wiersz też należy do definicji makra. Dodatkowo, dosyć często blok instrukcji w makrze nie jest tworzony za pomocą zwykłej pary nawiasów klamrowych, ale z użyciem pętli `do…while()`, Wszystkie instrukcje makra są umieszczane w takiej pętli, po której nie występuje średnik, a w nawiasach okrągłych występujących po słowie kluczowym `while` wpisana jest liczba `0`. Taki zapis pętli gwarantuje, że wykona się ona tylko raz. Użycie go pozwala na umieszczenie średnika za rozwinięciem makra, jeśli kompilator oczekuje, że ten średnik powinien się tam znaleźć.

Makra preprocesora

Przykład

```
#include<stdio.h>

#define SWAP(A,B) do {\
    int tmp;\
    tmp = (A);\
    (A) = (B);\
    (B) = tmp; \
} while(0)

#define PRINT(A,B,C) printf("%s: %d, %s: %d, %s: %d\n",#A,(A),#B,(B),#C,(C))

int main(void)
{
    int a=2, b=3, c = 4;
    if(a>b)
        SWAP(a,b);
    else
        SWAP(a,c);
    PRINT(a,b,c);
    return 0;
}
```

Makra preprocesora

Komentarz

Zastosowanie pętli `do...while(0)` w kodzie makra `SWAP` pozwoliło umieścić średnik po jego rozwinięciu w instrukcji warunkowej przed słowem kluczowym `else`. Moglibyśmy z niego zrezygnować i zastosować w makrze zwykłe nawiasy klamrowe, jednak zapis taki byłby mniej czytelny. W programie zostało zdefiniowane dodatkowe makro o nazwie `PRINT`, które służy do wypisania wartości zmiennych na ekran. Proszę zwrócić uwagę, na sposób, w jaki została użyta w tym makrze funkcja `printf()`. Zapis `#A`, oznacza, że wyrażenie podstawione za ten parametr (w programie jest to zmienna a) zostanie zamienione na ciąg znaków. Oznacza to, że w komunikacie wartości zmiennych zostaną poprzedzone ich nazwami. Preprocesor posiada jeszcze inny operator, który zapisywany jest następująco: ##. Służy on do łączenia ze sobą dwóch łańcuchów. Sposób jego użycia jest zaprezentowany na następnym slajdzie.

Makra preprocesora

Przykład

```
#include<stdio.h>

#define VARIABLE(SCOPE) int SCOPE##_variable

VARIABLE(global);

int main(void)
{
    VARIABLE(local) = 6;
    printf("global_variable: %d, local_variable: %d\n",
                                    global_variable, local_variable);
    return 0;
}
```

Makra preprocesora

Komentarz

W programie makro `VARIABLE` zostało użyte do zadeklarowania i zdefiniowania dwóch zmiennych o nazwach odpowiednio `global_variable` oraz `local_variable` typu `int`. W tym drugim przypadku zmiennej nadano także wartość początkową. Nie jest to czytelny zapis, ale pokazuje zastosowanie operatora ##. Definicję makra można usunąć z całości lub części kodu programu za pomocą dyrektywy `#undef`, po której powinna wystąpić nazwa makra. Od miejsca jej wystąpienia nie będzie dostępne makro o podanej nazwie.

Asercje

Makra są dosyć często stosowane do znajdowania błędów w programie lub do weryfikacji jego poprawności. W języku C, w pliku nagłówkowym `assert.h` zdefiniowano makro o nazwie `assert`, które jest realizacją koncepcji *asercji* lub *niezmienników* zaproponowanych przez Roberta Floyda. Niezmiennik jest to wyrażenie, które zawsze powinno być prawdziwe. Aby dowieść poprawności pewnych fragmentów kodu bada się wartości niezmienników np. przed wykonaniem i po zakończeniu pętli lub przed wywołaniem i po zakończeniu funkcji. Jeśli ich wartości nie są poprawne, to znaczy, że w programie jest defekt, który trzeba usunąć. Makro `assert` przyjmuje jeden argument, którym jest wyrażenie będące niezmiennikiem. Jeśli jest ono fałszywe, to `assert` przerywa działanie programu i wypisuje na ekranie stosowny komunikat. Następny slajd zawiera przykład użycia takiego makra.

Asercje

Przykład

```c
#include<stdio.h>
#include<assert.h>

int main(void)
{
    int i;
    for(i=1;i<10;i++) {
        assert(i<=5);
        printf("%d%%%d=%d ",5,i,5%i);
    }
    return 0;
}
```

Asercje

Przykład

W zamieszczonym na poprzednim slajdzie programie użyto makra `assert` celem sprawdzenia, czy dzielnik używany do wyznaczenia reszty z dzielenia nie jest większy od dzielnej. Jeśli makro wykryje, że tak nie jest, to przerwie działanie programu i wypisze komunikat, w którym wierszu wystąpił błąd. Proszę zwrócić uwagę także na funkcję `printf()`. Aby wypisać za jej pomocą znak % na ekranie należy poprzedzić go takim samym znakiem. Programiści dosyć często wyłączają asercje przed oddaniem oprogramowania do wdrożenia. Choć jest to dosyć kontrowersyjne działanie, to warto wiedzieć jak to zrobić. Wystarczy zdefiniować puste makro o nazwie `NDEBUG` w kodzie źródłowym programu lub jako opcję wywołania kompilatora. W tym pierwszym przypadku należy to zrobić przed włączeniem pliku nagłówkowego `assert.h`. Przykład na następnym slajdzie pokazuje jak to należy zrobić.

Asercje

Przykład

```c
#include<stdio.h>
#define NDEBUG
#include<assert.h>

int main(void)
{
    int i;
    for(i=1;i<10;i++) {
        assert(i<=5);
        printf("%d%%%d=%d ",5,i,5%i);
    }
    return 0;
}
```

Makra w debugowaniu

W języku C zostały zdefiniowane również inne makra, które mogą być pomocne w znajdowaniu defektów w kodzie źródłowym programu. Nie trzeba włączać żadnego pliku nagłówkowego, aby były one dostępne. Oto niektóre z nich:

- `__DATE__` zamieniane jest na datę w momencie kompilacji,
- `__TIME__` zamieniane jest na czas w momencie kompilacji,
- `__FILE__` zamieniane jest na nazwę kompilowanego pliku,
- `__LINE__` zamieniane jest na numer wiersza w kompilowanym pliku.

Kompilacja warunkowa

Preprocesor posiada dyrektywy, które pozwalają na sterowanie kompilacją, tj. określenie, kiedy i czy w ogóle dany fragment kodu źródłowego powinien zostać skompilowany. Widzieliśmy ich zastosowanie w przypadku wartownika pliku nagłówkowego. W tabeli zaprezentowane są niektóre z dodatkowych dyrektyw preprocesora, które służą do sterowania kompilacją.

`#ifdef`	Działa podobnie do `#ifndef`. Jeśli znacznik występujący za nią został wcześniej zdefiniowany, to preprocesor uwzględnia instrukcje występujące za nią, a jeśli nie, to je pomija.
`#else`	Działa podobnie do `else` w instrukcji warunkowej.
`#elif`	Stanowi połączenie dyrektyw `#else` i `#if`.

Przykłady użycia kompilacji warunkowej pokazano na następnym slajdzie.

Kompilacja warunkowa

Przykład

```c
#include<stdio.h>
#include<assert.h>

#ifndef NDEBUG
#define PRINT_VERBOSE(X) printf("Zmienna %s ma następującą wartość: %d\n",#X,(X))
#else
#define PRINT_VERBOSE(X)
#endif

int main(void)
{
    int i;
    for(i=1;i<10;i++) {
        PRINT_VERBOSE(i);
        assert(i<=5);
        printf("%d%%%d=%d ",5,i,5%i);
    }
    return 0;
}
```

Kompilacja warunkowa

Komentarz

W programie zostało zdefiniowane makro, PRINT_VERBOSE, które wypisuje komunikat informujący o wartości zmiennej. Jest to pomocna informacja, jeśli szukamy miejsca w kodzie źródłowym, gdzie występuje defekt. Jednakże, w normalnym trybie pracy programu nie jest ona potrzebna, a wręcz staje się zbędna. Dlatego, gdy zostanie zdefiniowany znacznik NDEBUG, ten sam, który jest używany dla makra `assert`, to definicja makra PRINT_VERBOSE stanie się pusta. Przykład, kiedy tak się dzieje pokazano na następnym slajdzie.

Kompilacja warunkowa

Przykład

```
#include<stdio.h>
#define NDEBUG
#include<assert.h>

#ifndef NDEBUG
#define PRINT_VERBOSE(X) printf("Zmienna %s ma następującą wartość: %d\n",#X,(X))
#else
#define PRINT_VERBOSE(X)
#endif

int main(void)
{
    int i;
    for(i=1;i<10;i++) {
        PRINT_VERBOSE(i);
        assert(i<=5);
        printf("%d%%%d=%d ",5,i,5%i);
    }
    return 0;
}
```

Zmienna liczba argumentów funkcji

Język C pozwala na tworzenie funkcji, które przyjmują zmienną liczbę argumentów wywołania. Przykładem podprogramu, który jest zrealizowany w ten sposób jest funkcja `printf()`. Obsługą listy takich argumentów zajmują się makra preprocesora, których definicje są umieszczone w pliku nagłówkowym `stdarg.h`. Aby funkcja mogła obsługiwać zmienną liczbę argumentów musi być odpowiednio zdefiniowana. Na jej liście parametrów musi występować przynajmniej jeden zwykły parametr. Jako ostatnia pozycja na takiej liście powinny występować trzy kropki (...), które są informacją dla kompilatora, że w ich miejsce może być podstawiona dowolna liczba argumentów wywołania. Parametr poprzedzający te kropki nie może być parametrem tablicowym.

Zmienna liczba argumentów funkcji

Dostęp do dodatkowych argumentów wywołania umożliwiają cztery makra. Pierwsze z nich to `va_start`, które przyjmuje dwa argumenty. Pierwszym jest zmienna typu `va_list`. Musi być ona wcześniej zadeklarowana (i zdefiniowana). Ta zmienna jest listą dodatkowych argumentów wywołania funkcji. Drugim argumentem opisywanego marka jest nazwa parametru poprzedzającego na liście parametrów funkcji trzy kropki. Zadaniem makra jest inicjalizacja zmiennej `va_list`. Makro `va_arg` zwraca wartość argumentu funkcji, którego typ został mu przekazany jako jego drugi argument. Pierwszym argumentem tego makra jest zmienna typu `va_list`. Makro `va_end` przyjmuje jeden argument, którym jest lista dodatkowych argumentów funkcji i sygnalizuje zakończenie korzystania z listy przez funkcję. Makro `va_copy` służy do kopiowania listy dodatkowych argumentów. Przyjmuje ono dwa własne argumenty, które są zmiennymi typu `va_list`. Wartość drugiego jest kopiowana do pierwszego.

Zmienna liczba argumentów funkcji

Przykład

```
#include<stdio.h>
#include<stdarg.h>

double average(unsigned int counter, ...)
{
    va_list arguments_list;
    int next=0, sum=0, i=counter;

    if(counter==0)
        return 0.0;

    va_start(arguments_list, counter);
    while(i--) {
        next = va_arg(arguments_list,int);
        sum += next;
    }
    va_end(arguments_list);
    return (double)sum/counter;
}
```

Zmienna liczba argumentów funkcji

Przykład

```
int main(void)
{
    double result = average(5,1,2,3,4,5);
    printf("Średnia liczb: %.2f\n",result);
    return 0;
}
```

Zmienna liczba argumentów funkcji

Komentarz do przyładu

Funkcja `average()` w zaprezentowanym na dwóch poprzednich slajdzach przykładowym programie liczy średnią przekazanych jej jako argumenty wywołania liczb całkowitych. Tylko pierwszy argument tej funkcji jest przekazywany przez znany parametr. Jego wartość określa ile będzie dodatkowych argumentów wywołania tej funkcji. Te argumenty nie muszą być tego samego typu, choć takie właśnie rozwiązanie zaprezentowano w przykładzie.

Funkcje inline

Funkcje `inline`, nazywane także funkcjami wplatanymi, są pewną alternatywą dla makr preprocesora. Podobnie jak one funkcje te nie są wywoływane, ale rozwijane w miejscu użycia. Tym razem czynność tę wykonuje jednak kompilator, a nie preprocesor, co oznacza, że sprawdzana jest poprawność takiego rozwinięcia. Zatem funkcja `inline` zachowuje się jak zwykła funkcja, z dokładnością do tego, że jej kod jest wklejany w kodzie w miejscu jej użycia. To powoduje oszczędność pamięci przeznaczonej na stos (nie trzeba tworzyć rekordu aktywacji dla tej funkcji) i przyspieszenie jej działania (funkcja nie jest wywoływana, tylko jest wykonywany kod w niej zawarty), kosztem dłuższego czasu kompilacji i większej objętości pliku z kodem wynikowym. Należy zaznaczyć, że standard języka C nie wymusza na kompilatorach opisanego traktowania funkcji `inline`, co oznacza, że niektóre z nich mogą traktować te funkcje tak samo, jak każde inne, lub stosować inne, bardziej ograniczone zabiegi celem skrócenia czasu ich wywoływania. Aby zwykłą funkcję uczynić funkcją `inline`, należy na początku jej nagłówka dodać słowa `static inline`. Kod źródłowy programu zawierającego taką funkcję został przedstawiony na następnym slajdzie.

Funkcje inline

Przykład

```c
#include<stdio.h>

static inline void swap(int *first, int *second)
{
    int tmp = *first;
    *first = *second;
    *second = tmp;
}

int main(void)
{
    int a = 1, b = 2;
    swap(&a,&b);
    printf("a: %d, b: %d",a,b);
    return 0;
}
```

Podziękowania

Składam podziękowania dla dra inż. Grzegorza Łukawskiego i mgra inż. Leszka Ciopińskiego za udostępnienie materiałów, których fragmenty zostały wykorzystane w tym wykładzie.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę.