1
PREPROCESOR

Gdy się nie wie, co się robi,
to dzieją się takie rzeczy,
że się nie wie, co się dzieje ;-).
znana prawda programistyczna

Poznawanie bardziej zaawansowanych cech języka C++ zaczniemy od czegoś, co pochodzi jeszcze z czasów jego poprzednika, czyli C. Podobnie jak wskaźniki, preprocesor nie pojawił się wraz z dwoma plusami w nazwie języka i programowaniem zorientowanym obiektowo, lecz był obecny od jego samych początków.

W przypadku wskaźników trzeba jednak powiedzieć, że są one także i teraz niezbędne do efektywnego i poprawnego konstruowania aplikacji. Natomiast o proceprocesorze niewielu ma tak pochlebne zdanie: według sporej części programistów, stał się on prawie zupełnie niepotrzebny wraz z wprowadzeniem do C++ takich elementów jak funkcje *inline* oraz szablony. Poza tym uważa się powszechnie, że częste i intensywne używanie tego narzędzia pogarsza czytelność kodu.
W tym rozdziale będę musiał odpowiedzieć jakoś na te opinie. Nie da się ukryć, że niektóre z nich są słuszne: rzeczywiście, era świetności preprocesora jest już dawno za nami. Zgadza się, nadmierne i nieuzasadnione wykorzystywanie tego mechanizmu może przynieść więcej szkody niż pożytku. Tym bardziej jednak powinieneś wiedzieć jak najwięcej na temat tego elementu języka, aby móc stosować go poprawnie. Od korzystania z niego nie można bowiem uciec. Choć może nie zdawałeś sobie z tego sprawy, lecz korzystałeś z niego w **każdym** napisanym dotąd programie w C++! Wspomnij sobie choćby dyrektywę `#include`…

Dotąd jednak zadowalałeś się lakonicznym stwierdzeniem, iż tak po prostu „trzeba”. Lekturą tego rozdziału masz szansę to zmienić. Teraz bowiem omówimy sobie zagadnienie preprocesora w całości, od początku do końca i od środka :)

Pomocnik kompilatora

Rozpocząć wypadałoby od przedstawienia głównego bohatera naszej opowieści. Czym jest więc preprocesor?…

Preprocesor to specjalny mechanizm języka, który przetwarza **tekst programu** jeszcze **przed jego kompilacją**.

To jakby przedsionek właściwego procesu kompilacji programu. Preprocesor przygotowuje kod tak, aby kompilator mógł go skompilować zgodnie z życzeniem programisty. Bardzo często uwalnia on też od konieczności powtarzania często występujących i potrzebnych fragmentów kodu, jak na przykład deklaracji funkcji.

Kiedy wiemy już mniej więcej, czym jest preprocesor, przyjrzymy się wykonywanej przez niego pracy. Dowiemy się po prostu, co on robi.

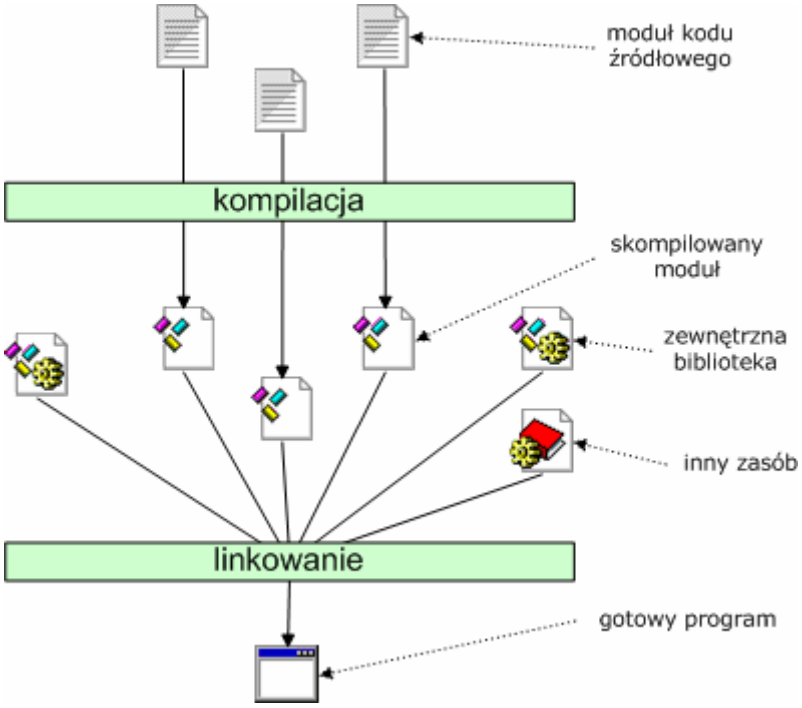
Gdzie on jest…?

Obecność w procesie budowania aplikacji nie jest taka oczywista. Całkiem duża liczba języków radzi sobie, nie posiadając w ogóle narzędzia tego typu. Również cel jego istnienia wydaje się niezbyt klarowny: dlaczego kod naszych programów miałby wymagać przed kompilacją jakichś przeróbek?…

Tę drugą wątpliwość wyjaśnią kolejne podrozdziały, opisujące możliwości i polecenia preprocesora. Obecnie zaś określimy sobie jego miejsce w procesie tworzenia wynikowego programu.

Zwyczajowy przebieg budowania programu

W języku programowania nieposiadającym preprocesora generowanie docelowego pliku z programem przebiega, jak wiemy, w dwóch etapach.
Pierwszym jest **kompilacja**, w trakcie której kompilator przetwarza kod źródłowy aplikacji i produkuje skompilowany kod maszynowy, zapisany w osobnych plikach. Każdy taki plik - wynik pracy kompilatora - odpowiada jednemu modułowi kodu źródłowego.
W drugiem etapie następuje **linkowanie** skompilowanych wcześniej modułów oraz ewentualnych innych kodów, niezbędnych do działania programu. W wyniku tego procesu powstaje gotowy program.



Schemat 36. Najprostszy proces budowania programu z kodu źródłowego

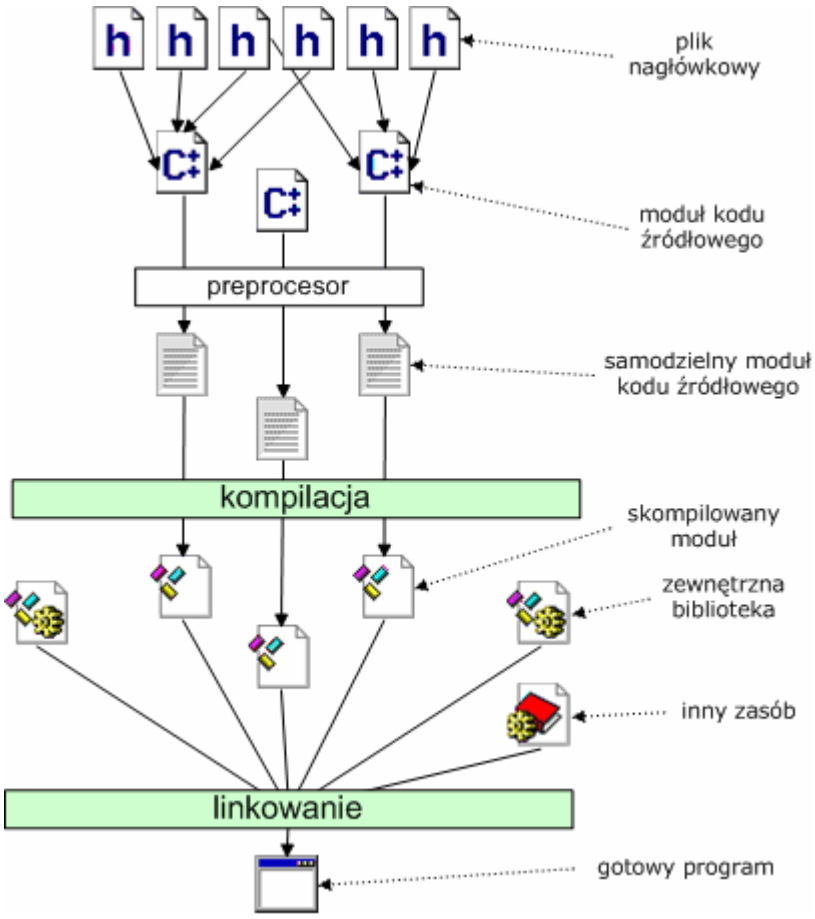
Przy takim modelu kompilacji zawartość każdego modułu musi wystarczać do jego samodzielnej kompilacji, niezależnej od innych modułów. W przypadku języków z rodziny C oznacza to, że każdy moduł musi zawierać deklaracje używanych funkcji oraz definicje klas, których obiekty tworzy i z których korzysta.
Gdyby zadanie dołączania tych wszystkich deklaracji spoczywało na programiście, to byłoby to dla niego niezmiernie uciążliwe. Pliki z kodem zostały ponadto rozdęte do nieprzyzwoitych rozmiarów, a i tak większość zawartych weń informacji przydawałyby się tylko przez chwilę. Przez tą chwilę, którą zajmuje kompilacja modułu.

Nic więc dziwnego, że aby zapobiec podobnym irracjonalnym wymaganiom wprowadzono mechanizm preprocesora.

Dodajemy preprocesor

Ujawnił się nam pierwszy cel istnienia preprocesora: w języku C(++) służy on do łączenia w jedną całość modułów kodu wraz z deklaracjami, które są niezbędne do działania tegoż kodu. A skąd brane są te deklaracje?…
Oczywiście - z plików nagłówkowych. Zawierają one przecież prototypy funkcji i definicje klas, z jakich można korzystać, jeżeli dołączy się dany nagłówek do swojego modułu.

Jednak kompilator nic **nie wie** o plikach nagłówkowych. On tylko oczekuje, że zostaną mu podane pliki z kodem źródłowym, do którego będą się zaliczały także deklaracje pewnych zewnętrznych elementów - nieobecnych w danym module. Kompilator potrzebuje tylko ich określenia „z wierzchu", bez wnikania w implementację, gdyż ta może znajdować się w innych modułach lub nawet innych bibliotekach i staje się ważna dopiero przy linkowaniu. Nie jest już ona sprawą kompilatora - on żąda tylko tych informacji, które są mu potrzebne do kompilacji.
Niezbędne deklaracje powinny się znaleźć na początku każdego modułu. Trudno jednak oczekiwać, żebyśmy wpisywali je ręcznie w **każdym** module, który ich wymaga. Byłoby to niezmiernie uciążliwe, więc wymyślono w tym celu pliki nagłówkowe… i preprocesor. Jego zadaniem jest tutaj połączenie napisanych przez nas modułów oraz plików nagłówkowych w pliki z kodem, które mogą mogą być bez przeszkód przetworzone przez kompilator.



Schemat 37. Budowanie programu C++ z udziałem preprocesora

Skąd preprocesor wie, jak ma to zrobić?… Otóż, mówimy o tym wyraźnie, stosując dyrektywę `#include`. W miejscu jej pojawienia się zostaje po prostu wstawiona treść odpowiedniego pliku nagłówkowego.

Włączanie nagłówków nie jest jednak jedynym działaniem podejmowanym przez preprocesor. Gdyby tak było, to przecież nie poświęcalibyśmy mu całego rozdziału :) Jest wręcz przeciwnie: dołączanie plików to tylko jedna z czynności, jaką możemy zlecić temu mechanizmowi - jedna z wielu czynności…
Wszystkie zadania preprocesora są różnorodne, ale mają też kilka cech wspólnych. Przyjrzyjmy się im w tym momencie.

Działanie preprocesora

Komendy, jakie wydajemy preprocesorowi, różnią się od normalnych instrukcji języka programowania. Także sposób, w jaki preprocesor traktuje kod źródłowy, jest zupełnie inny.

Dyrektywy

Polecenie dla preprocesora nazywamy jego **dyrektywą** (ang. *directive*). Jest to specjalna linijka kodu źródłowego, rozpoczynająca się od znaku `#` (*hash*), zwanego płotkiem[97]:

```
#
```

Na nim też może się zakończyć - wtedy mamy do czynienia z dyrektywą pustą. Jest ona ignorowana przez preprocesor i nie wykonuje żadnych czynności.

Bardziej praktyczne są inne dyrektywy, których nazwy piszemy zaraz za znakiem `#`. Nie oddzielamy ich żadnymi spacjami, więc w praktyce płotek staje się częścią ich nazw. Mówi się więc o instrukcjach `#include`, `#define`, `#pragma` i innych, gdyż w takiej formie zapisujemy je w kodzie.
Dalsza część dyrektywy zależy już od jej rodzaju. Różne „parametry” dyrektyw poznamy, gdy zajmiemy się szczegółowo każdą z nich.

Bez średnika

Jest bardzo ważne, aby zapamiętać, że:

Dyrektywy preprocesora kończą się zawsze przejściem do następnego wiersza.

Innymi słowy, jeżeli preprocesor napotka w swojej dyrektywie na znak końca linijki (nie widać go w kodzie, ale jest on dodawany po każdym wciśnięciu *Enter*), to uznaje go także za koniec dyrektywy. Nie ma potrzeby wpisywania średnika na zakończenie instrukcji. Więcej nawet: nie powinno się go wpisywać! Zostanie on bowiem uznany za część dyrektywy, co w zależności od jej rodzaju może powodować różne niepożądane efekty. Kończą się one zwykle błędami kompilacji.

Zapamiętaj zatem zalecenie:

Nie kończ dyrektyw preprocesora **średnikiem**. Nie są to przecież instrukcje języka programowania, lecz polecenia dla modułu wspomagającego kompilator.

[97] Przed *hashem* mogą znajdować się wyłącznie tzw. białe znaki, czyli spacje lub tabulatory. Zwykle nie znajduje się nic.

Można natomiast kończyć dyrektywę komentarzem, opisującym jej działanie. Kiedyś wiele kompilatorów miało z tym kłopoty, ale obecnie wszystkie liczące się produkty potrafią radzić sobie z komentarzami na końcu dyrektyw preprocesora.

Ciekawostka: sekwencje trójznakowe

Istnieje jeszcze jedna, bardzo rzadka dzisiaj sytuacja, gdy preprocesor zostaje wezwany do akcji. Jest to jedyny przypadek, kiedy jego praca jest niezwiązana z dyrektywami obecnymi w kodzie.
Chodzi o tak zwane **sekwencje trójznakowe** (ang. *trigraphs*). Cóż to takiego?…

W każdym długo i szeroko wykorzystywanym produkcie pewne funkcje mogą być po pewnym czasie uznane za przestarzałe i przestać być wykorzystywane. Jeżeli mimo to są one zachowywane w kolejnych wersjach, to zyskują słuszne miano skamieniałości (ang. *fossils*).
Język C++ zawiera kilka takich zmumifikowanych konstrukcji, odziedziczonych po swoim poprzedniku. Jedną z nich jest na przykład możliwość wpisywania do kodu liczb w systemie ósemkowym (oktalnym), poprzedzając je zerem (np. `042` to dziesiętnie `34`). Obecnie jest to całkowicie niepotrzebne, jako że współczesny programista nie odniesie żadnej korzyści z wykorzystania tego systemu liczbowego. W architekturze komputerów został on bowiem całkowicie zastąpiony przez szesnastkowy (heksadecymalny) sposób liczenia. Ten jest na szczęście także obsługiwany przez C++[98], natomiast zachowana możliwość użycia systemu oktalnego stała się raczej niedogodnością niż plusem języka. Łatwo przecież omyłkowo wpisać zero przed liczbą dziesiętną i zastanawiać się nad powstałym błędem…

Inną skamieniałością są właśnie sekwencje trójznakowe. To specjalne złożenia dwóch znaków zapytania (`??`) oraz innego trzeciego znaku, które razem „udają” symbol ważny dla języka C++. Preprocesor zastępuje te sekwencje docelowym znakiem, postępując według tej tabelki:

trójznak	***symbol***
`??=`	`#`
`??/`	`\`
`??-`	`~`
`??'`	`^`
`??!`	`|`
`??(`	`[`
`??)`	`]`
`??<`	`{`
`??>`	`}`

Tabela 13. Sekwencje trójznakowe w C++

Twórca języka C++, Bjarne Stroustroup, wprowadził do niego sekwencje trójznakowe z powodu swojej… klawiatury. W wielu duńskich układach klawiszy zamiast przydatnych symboli z prawej kolumny tabeli widniały bowiem znaki typu å, Æ czy Å. Aby umożliwić swoim rodakom programowanie w stworzonym języku, Stroustroup zdecydował się na ten zabieg.
Dzisiaj obecność trójznaków nie jest taka ważna, bo powszechnie występują na całym świecie klawiatury typu Sholesa, które zawierają potrzebne w C++ znaki. Moglibyśmy więc o nich zapomnieć, ale…

[98] Aby zapisać liczbę w systemie szesnastkowym, należy ją poprzedzić sekwencją `0x` lub `0X`. Tak więc `0xFF` to dziesiętnie `255`.

No właśnie, jest pewien problem. Z niewiadomych przyczyn jest często tak, że nieużywana funkcja prędzej czy później daje o sobie znać niczym przeterminowana konserwa. Prawie zawsze też nie jest to zbyt przyjemne.
Kłopot polega na tym, że jedna z sekwencji - `??!` - może być użyta w sytuacji wcale odmiennej od założonego zastępowania znaku |. Popatrzmy na ten kod:

```
std::cout << "Co mowisz??!";
```

Nie wypisze on wcale stanowczej prośby o powtórzenie wypowiedzi, lecz napis `"Co mowisz|"`. Trójznak `??!` został bowiem zastąpiony przez `|`.
Można tego uniknąć, stosując jedną z tzw. **sekwencji ucieczki** (**unikowych**, ang. *escape sequences*) zamiast znaków zapytania. Poprawiony kod będzie wyglądał tak:

```
std::cout << "Co mowisz\?\?!";
```

Podobną niespodziankę możemy też sobie sprawić, gdy podczas wpisywania trzech znaków zapytania za wcześnie zwolnimy klawisz *Shift*. Powstanie nam wtedy coś takiego:

```
std::cout << "Co??/";
```

Taka sytuacja jest znacznie perfidniejsza, bowiem trójznak `??/` zostanie zastąpiony przez pojedynczy znak `\` (backslash). Doprowadzi to do powstania **niekompletnego** napisu `"Co\"`. Niekompletnego, bo występuje tu sekwencja unikowa `\"`, zastępująca cudzysłów. Znak cudzysłowu, który tu widzimy, nie będzie wcale oznaczał końca napisu, lecz jego część. Kompilator będzie zaś oczekiwał, że właściwy cudzysłów kończący znajduje się gdzieś dalej, w tej samej linijce kodu. Nie napotka go oczywiście, a to oznacza dla nas kłopoty…

Musimy więc pamiętać, aby bacznie przyglądać się każdemu wystąpieniu dwóch znaków zapytania w kodzie C++. Takie skamieniałe okazy nawet po wielu latach mogą dotkliwie kąsać nieostrożnego programistę.

Preprocesor a reszta kodu

Nadmieniłem wcześniej, że dyrektywy preprocesora różnią się od normalnych instrukcji języka C++ - choćby tym, że na ich końcu nie stawiamy średnika. Ale nie jest to jeszcze cała prawda.

Najważniejsze jest to, jak preprocesor obchodzi się kodem źródłowym programu. Jego podejście jest odmienne od kompilatora. Tak naprawdę to preprocesor w zasadzie „nie wie", że przetwarzany przez niego tekst jest programem! Wiedza ta nie jest mu do niczego potrzeba, gdyż traktuje on kod jak **każdy inny tekst**. Dla preprocesora nie ma różnicy, czy pracuje na prostym programie konsolowym, zaawansowanej aplikacji okienkowej, czy nawet (hipotetycznie) na siódmej księdze *Pana Tadeusza*.
Możnaby więc powiedzieć, że preprocesor jest po prostu głupi - gdyby nie to, że bardzo dobrze radzi sobie ze swoim zadaniem. A jest nim przetwarzanie **tekstu programu** w taki sposób, aby ułatwić życie programiście. Dzięki preprocesorowi można bowiem automatycznie wykonać operacje, które bez niego zajmowałyby mnóstwo czasu i słusznie wydawały się jego kompletną, frustrującą stratą.

Jak zwykle jednak trzeba wtrącić jakieś „ale" :) Całkowita niewiedza preprocesora na temat podmiotu jego działań może i jest błogosławieństwem dla niego samego, lecz stosunkowo łatwo może stać się przyczyną błędów kompilacji. Dotyczy to w szczególności jednego z aspektów wykorzystania preprocesora - makr.

Makra

Makro (ang. *macro*) jest instrukcją dla preprocesora, pozwalającą dokonywać zastąpienia pewnego wyrażenia innym. Działa ona trochę jak funkcja *Znajdź i zamień* w edytorach tekstu, z tym że proces zamiany dokonuje się wyłącznie przed kompilacją i nie jest trwały. Pliki z kodem źródłowym nie są fizycznie modyfikowane, lecz tylko zmieniona ich postać trafia do kompilatora.
Makra w C++ (zwane aczkolwiek częściej makrami C) potrafią być też nieco bardziej wyrafinowane i dokonywać złożonych, sparametryzowanych operacji zamiany tekstu. Takie makra przypominają funkcje i zajmiemy się nimi nieco dalej.

Definicja makra odbywa się przy pomocy dyrektywy `#define`:

```
#define odwołanie tekst
```

Najogólniej mówiąc, daje to taki efekt, iż każde wystąpienie *odwołania* w kodzie programu powoduje jego zastąpienie przez *tekst*. Szczegóły tego procesu zależą od tego, czy nasze makro jest proste - udające stałą - czy może bardziej skomplikowane - udające funkcję. Osobno zajmiemy się każdym z tych dwóch przypadków.

Do pary z `#define` mamy jeszcze dyrektywę `#undef`:

```
#undef odwołanie
```

Anuluje ona poprzednią definicję makra, pozwalając na przykład na jego ponowne zdefiniowanie. Makro w swej aktualnej postaci jest więc dostępne od miejsca zdefiniowania do wystąpienia `#undef` lub końca pliku.

Proste makra

W prostej postaci dyrektywa `#define` wygląda tak:

```
#define wyraz [zastępczy_ciąg_znaków]
```

Powoduje ona, że w pliku wysłanym do kompilacji każde samodzielne[99] wystąpienie *wyrazu* zostanie zastąpione przez podany *zastępczy_ciąg_znaków*. Mówimy o tym, że **makro zostanie rozwinięte**. W *wyrazie* mogą wystąpić tylko znaki dozwolone w nazwach języka C++, a więc litery, cyfry i znak podkreślenia. Nie może on zawierać spacji ani innych białych znaków, gdyż w przeciwnym razie jego część zostanie zinterpretowana jako treść makra (*zastępczy_ciąg_znaków*), a nie jako jego nazwa. Treść makra, czyli *zastępczy_ciąg_znaków*, może natomiast zawierać białe znaki. Może także nie zawierać znaków - nie tylko białych, ale w ogóle żadnych. Wtedy każde wystapienie *wyrazu* zostanie usunięte przez preprocesor z pliku źródłowego.

Definiowianie prostych makr

Jak wygląda przykład opisanego wyżej użycia `#define`? Popatrzmy:

```
#define SIEDEM 7

// (tutaj trochę kodu programu...)

std::cout << SIEDEM << "elementow tablicy" << std::endl;;
```

[99] Samodzielne - to znaczy jako odrębne słowo (token).

```
int aTablica[SIEDEM];

for (unsigned i = 0; i < SIEDEM; ++i)
   std::cout << aTablica[i] << std::endl;

std::cout << "Wypisalem SIEDEM elementow tablicy";
```

Nie możemy tego wprawdzie zobaczyć, ale uwierzmy (lub sprawdźmy empirycznie poprzez kompilację), że preprocesor zamieni powyższy kod na coś takiego:

```
std::cout << 7 << "elementow tablicy" << std::endl;;
int aTablica[7];

for (unsigned i = 0; i < 7; ++i)
   std::cout << aTablica[i] << std::endl;

std::cout << "Wypisalem SIEDEM elementow tablicy";
```

Zauważmy koniecznie, że:

Preprocesor **nie dokonuje zastępowania** nazw makr **wewnątrz napisów**.

Jest to uzasadnione, bo wewnątrz łańcucha nazwa może występować w zupełnie innym znaczeniu. Zwykle więc nie chcemy, aby została ona zastąpiona przez rozwinięcie makra. Jeżeli jednak życzymy sobie tego, musimy potraktować makro jak zmienną, czyli na przykład tak:

```
std::cout << "Wypisalem " << SIEDEM << " elementow tablicy";
```

Poza łańcuchami znaków makro jest bowiem wystawione na działanie preprocesora.

Zgodnie z przyjętą powszechnie konwencją, nazwy makr piszemy wielkimi literami. Nie jest to rzecz jasna obowiązkowe, ale poprawia czytelność kodu.

Zastępowanie większych fragmentów kodu

Zamiast jednej liczby czy innego wyrażenia, jako treść makra możemy też podać instrukcję. Może to nam zaoszczędzić pisania. Przykładowo, jeżeli przed wyjściem z funkcji musimy zawsze wyzerować jakąś zmienną globalną, to możemy napisać sobie odpowiednie makro:

```
#define ZAKONCZ      { g_nZmienna = 0; return; }
```

Jest to przydatne, jeśli w kodzie funkcji mamy wiele miejsc, które mogą wymagać jej zakończenia. Każdorazowe ręczne wpisywanie tego kodu byłoby więc uciążliwe, zaś z pomocą makra staje się proste.
Przypomnijmy jeszcze, jak to działa. Jeżeli mamy taką oto funkcję:

```
void Funkcja()
{
   // ...

   if (!DrugaFunkcja())    ZAKONCZ;
   // ...
   if (!TrzeciaFunkcja())  ZAKONCZ;
   // ...
   if (CosSieStalo())      ZAKONCZ;
   // ...
}
```

to preprocesor zamieni ją na coś takiego:

```
void Funkcja()
{
   // ...

   if (!DrugaFunkcja())    { g_nZmienna = 0; return; };
   // ...
   if (!TrzeciaFunkcja())  { g_nZmienna = 0; return; };
   // ...
   if (CosSieStalo())      { g_nZmienna = 0; return; };
   // ...
}
```

Wyodrębnienie kodu w postaci makra ma tę zaletę, że jeśli nazwa zmiennej `g_nZmienna` zmieni się (;D), to modyfikację poczynimy tylko w jednym miejscu - w definicji makra.

Spójrzmy jeszcze, iż treść makra ująłem w nawiasy klamrowe. Gdybym tego nie zrobił, to otrzymalibyśmy kod typu:

```
if (!DrugaFunkcja()) g_nZmienna = 0; return;;
```

Nie widać tego wyraźnie, ale kodem wykonywanym w razie prawdziwości warunku if jest tu tylko wyzerowanie zmiennej. Instrukcja `return` zostanie wykonana niezależnie od okoliczności, bo znajduje się poza blokiem warunkowym.
Przyzwoity kompilator powie nam o tym, bo obecność takiej zgubionej instrukcji powoduje zbędność całego dalszego kodu funkcji. Nie zawsze jednak korzystamy z makr zawierających `return`, zatem:

Zawsze umieszczajmy **treść makr w nawiasach**.

Jak się niedługo przekonamy, ta stanowcza sugestia dotyczy też makr typu stałych (jak `SIEDEM` z pierwszego przykładu), lecz w ich przypadku chodzi o nawiasy okrągłe.

Wątpliwości może budzić nadmiar średników w powyższych przykładach. Ponieważ jednak nie poprzedzają ich żadne instrukcje, więc dodatkowe średniki zostaną zignorowane przez kompilator. Akurat w tej sytuacji nie jest to problemem…

W kilku linijkach

Pisząc makra zastępujące całe połacie kodu, możemy je podzielić na kilka linijek. W tym celu korzystamy ze znaku `\` (backslash), np. w ten sposób:

```
#define WYPISZ_TABLICE      for (unsigned i = 0; i < 10; ++i)        \
                            {                                        \
                                 std::cout << i << "-ty element";    \
                                 std::cout << nTab[i] << std::endl;  \
                            }
```

Pamiętajmy, że to konieczne tylko dla dyrektyw preprocesora. W przypadku zwykłych instrukcji wiemy doskonale, że ich podział na linie jest całkowicie dowolny.

Makra korzystające z innych makr

Nic nie stoi na przeszkodzie, aby nasze makra korzystały z innych wcześniej zdefiniowanych makr:

```
#define PI 3.1415926535897932384
```

```
#define PROMIEN 10
#define OBWOD_KOLA (2 * PI * PROMIEN)
```

Mówiąc ściślej, to makra mogą korzystać ze wszystkich informacji dostępnych w czasie kompilacji programu, a więc np. operatora `sizeof`, typów wyliczeniowych lub stałych.

Pojedynek: makra kontra stałe

No właśnie - stałych… Większość przedstawionych tutaj makr pełni przecież taką samą rolę, jak stałe deklarowane słówkiem `const`. Czy obie konstrukcje są więc sobie równoważne?…

Nie. Stałe deklarowane przez `const` i „stałe” (makra) definiowane przez `#define` różnią się od siebie, i to znacznie. Te różnice dają przewagę obiektom `const` - powiedzmy tu sobie, dlaczego.

Makra nie są zmiennymi

Patrząc na ten tytuł pewnie się uśmiechasz. Oczywiście, że makra nie są zmiennymi - przecież to stałe… a raczej „stałe”. To jednak nie jest wcale takie oczywiste, bo z kolei stałe deklarowane przez `const` mają cechy zmiennych. Swego czasu mówiłem nawet na poły żartobliwie, iż te stałe są to zmienne, które są niezmienne. Makra `#define` takimi zmiennymi nie są, a przez to tracą ich cenne właściwości.
Jakie?… Zasięg, miejsce w pamięci i typ.

Zasięg

Brak zasięgu jest szczególnie dotkliwy. Makra mają wprawdzie zakres obowiązywania, wyznaczany przez dyrektywy `#define` i `#undef` (względnie koniec pliku), ale absolutnie nie jest to tożsame pojęcia.

Makro zdefiniowane - jak się zdaje - wewnątrz funkcji:

```
void Funkcja()
{
   #define STALA 1500.100900
}
```

nie jest wcale dostępne tylko wewnątrz niej. Z równym powodzeniem możemy z niego korzystać także w kodzie następującym dalej. Wszystko dlatego, że preprocesor nie zdaje sobie w ogóle sprawy z istnienia takiego czegoś jak „funkcje” czy „bloki kodu”, a już na pewno nie „zasięg zmiennych”. Nie jest zatem dziwne, że jego makra nie posiadają zasięgu.

Miejsce w pamięci i adres

Nazwy makr nie są znane kompilatorowi, ponieważ znikają one po przetworzeniu programu przez preprocesor. „Stałe” definiowane przez `#define` nie mogą zatem istnieć fizycznie w pamięci, bo za jej przydzielanie dla obiektów niedynamicznych odpowiedzialny jest wyłącznie kompilator. Makra nie zajmują miejsca w pamięci operacyjnej i nie możemy pobierać ich adresów. Byłoby to podobne do pobierania wskaźnika na liczbę `5`, czyli całkowicie bezsensowne i niedopuszczalne.

Ale chwileczkę… Brak możliwości pobrania wskaźnika łatwo można przetrawić, bo przecież nie robi się tego często. Nieobecność makr w pamięci ma natomiast oczywistą zaletę: nie zajmują jej swoimi wartościami. To chyba dobrze, prawda?
Tak, to dobrze. Ale jeszcze lepiej, że obiekty `const` także to potrafią. Każdy szanujący się kompilator nie będzie alokował pamięci dla stałej, jeżeli nie jest to potrzebne. Jeśli więc nie pobieramy adresu stałej, to będzie ona zachowywała się w identyczny sposób jak

makro - pod względem zerowego wykorzystania pamięci. Jednocześnie zachowa też pożądane cechy zmiennej. Mamy więc dwie pieczenie na jednym ogniu, a makra mogą się spalić… ze wstydu ;)

Typ

Makra nie mają też typów. „Jak to?!”, odpowiesz. „A czy `67` jest napisem, albo czy `"klawiatura"` jest liczbą? A przecież i te, i podobne wyrażenia mogą być treścią makr!” Faktycznie wyrażenia te mają swoje typy i mogą być interpretowane tylko w zgodzie z nimi. Ale jakie są to typy? `67` może być przecież równie dobrze uznana za wartość `int`, jak i `BYTE`, `unsigned`, nawet `float`. Z kolei napis jest formalnie typu `const char[]`, ale przecież możemy go przypisać do obiektu `std::string`. Poprzez występowanie niejawnych konwersji (powiemy sobie o nich w następnym rozdziale) sytuacja z typami nie jest więc taka prosta.
A makra dodatkowo ją komplikują, bo nie pozwalają na ustalenie typu stałej. Nasze `67` mogło być przecież docelowo typu `float`, ale „stała” zdefiniowana jako:

```
#define STALA 67
```

zostanie bez przeszkód przyjęta dla każdego typu liczbowego. O to nam chyba nie chodziło?!

Z tym problemem można sobie aczkolwiek poradzić, nie uciekając od `#define`. Pierwszym wyjściem jest jawne rzutowanie:

```
#define (float) 67
```

Chyba nieco lepsze jest dodanie do liczby odpowiedniej końcówki, umożliwiającej inną interpretację jej typu. Stosując te końcówki możemy zmienić typ wyrażenia wpisanego w kodzie. Oto jak zmienia się typ liczby 67, gdy dodamy jej różne sufiksy (nie są to wszystkie możliwości):

liczba	***typ***
`67`	`int`
`67u`	`unsigned int`
`67.0`	`double`
`67.0f`	`float`

Tabela 14. Typ stałej liczbowej w zależności od sposobu jej zapisu

Przewaga stałych `const` związana z typami objawia się najpełniej, gdy chodzi o tablice. Nie ma bowiem żadnych przeciwskazań, aby zadeklarować sobie tablicę wartości stałych:

```
const int STALE = { 1, 2, 3, 4 };
```

a potem odwoływać się do jej poszczególnych elementów. Podobne działanie jest całkowicie niemożliwe dla makr.

Efekty składniowe

Z wartościami stałymi definiowanymi jako makra związane też są pewne nieoczekiwane i trudne do przewidzenia efekty składniowe. Powoduje je fakt, iż działanie preprocesora jest operacją na zwykłym tekście, a kod przecież zwykłym tekstem nie jest…

Średnik

Podkreślałem na początku, że dyrektyw preprocesora, w tym i `#define`, nie należy kończyć średnikiem. Ale co by się stało, gdyby nie zastosować się do tego zalecenia?… Sprawdźmy. Zdefiniujmy na przykład takie oto makro:

```
#define DZIESIEC 10;       // uwaga, średnik!
```

Niby różnica jest niewielka, ale zaraz zobaczymy jak bardzo jest ona znacząca. Użyjmy teraz naszego makra, w jakimś wyrażeniu:

```
int nZmienna = 2 * DZIESIEC;
```

Działa? Tak… Preprocesor zamienia `DZIESIEC` na `10;`, co w sumie daje:

```
int nZmienna = 2 * 10;;
```

Dodatkowy średnik, jaki tu występuje, nie sprawia kłopotów, lecz łatwo może je wywołać. Wystarczy choćby przestawić kolejność czynników lub rozbudować wyrażenie - na przykład umieścić w nim wywołanie funkcji:

```
int nZmienna = abs(2 * DZIESIEC);
```

I tu zaczynają się kłopoty. Preprocesor wyprodukuje z powyższego wiersza kod:

```
int nZmienna = abs(2 * 10;);     // ups!
```

który z pewnością zostanie odrzucony przez każdy kompilator.

Słusznie jednak stwierdzisz, że takie czy podobne błędy (np. użycie `DZIESIEC` jako rozmiaru tablicy) są stosunkowo proste do wykrycia. Lecz przy używaniu makr nie zawsze tak jest: zaraz zobaczysz, że nietrudno dopuścić się pomyłek niewpływających na kompilację, ale wypływających na powierzchnię już w gotowym programie.

Nawiasy i priorytety operatorów

Popatrz na ten oto przykład:

```
#define SZEROKOSC 10
#define WYSOKOSC 20
#define POLE SZEROKOSC * WYSOKOSC
#define LUDNOSC 10000

std::cout << "Gestosc zaludnienia wynosi: " << LUDNOSC / POLE;
```

Powinien on wydrukować liczbę 50, prawda? No cóż, zobaczmy czy tak będzie naprawdę. Wyrażenie `LUDNOSC / POLE` zostanie rozwinięte przez preprocesor do:

```
LUDNOSC / SZEROKOSC * WYSOKOSC
```

czyli w konsekwencji do działań na liczbach:

```
10000 / 10 * 20
```

a to daje w wyniku:

```
1000 * 20
```

czyli ostatecznie:

```
20000          // ??? Coś jest nie tak!
```

Hmm… Pięćdziesiąt a dwadzieścia tysięcy to raczej duża różnica, znajdźmy więc błąd. Nie jest to trudne - tkwi on już w pierwszym kroku rozwijania makra:

```
LUDNOSC / SZEROKOSC * WYSOKOSC
```

Zgodnie z regułami kolejnościami działań, zwanych w programowaniu priorytetami operatorów, wpierw wykonywane jest tu dzielenie. To błąd - przecież najpierw powinniśmy obliczać wartość powierzchni, czyli iloczynu `SZEROKOSC * WYSOKOSC`. Należałoby zatem objąć go w nawiasy, i to najlepiej już przy definicji makra `POLE`:

```
#define POLE (SZEROKOSC * WYSOKOSC)
```

Całkiem nietrudno o tym zapomnieć. Jeszcze łatwiej przeoczyć fakt, że i `SZEROKOSC`, i `WYSOKOSC` mogą być także złożonymi wyrażeniami, więc również i one powinny posiadać własną parę nawiasów. Może nie być wiadome, czy w ich definicjach takie nawiasy występują, zatem przydałoby się wprowadzić je powyżej…

Mamy więc całkiem sporo niewiadomych podczas korzystania ze stałych-makr. A przecież wcale nie musimy rozstrzygać takich dylematów - zastosujmy po prostu stałe będące obiektami `const`:

```
const int SZEROKOSC = 10;
const int WYSOKOSC = 20;
const int POLE = SZEROKOSC * WYSOKOSC;
const int LUDNOSC = 10000;

std::cout << "Gestosc zaludnienia wynosi: " << LUDNOSC / POLE;
```

Teraz wszystko będzie dobrze. Ponieważ to inteligentny kompilator zajmuje się takimi stałymi (traktując je jak „niezmienne zmienne”), wartość wyrażenia `LUDNOSC / POLE` jest obliczana właściwie.

Dygresja: odpowiedź na pytanie o sens życia

Jak ciekawe skutki może wywoływać niewłaściwe użycie makr? Całkiem znamienne. Przypadkowo można na przykład poznać Najważniejszą Liczbę Wszechświata.

A tą liczbą jest… 42. Ów magiczny numer pochodzi z serii science-fiction *Autostopem przez Galaktykę* autorstwa Douglasa Adamsa. Tam też pada odpowiedź na Najważniejsze Pytanie o Życie, Uniwersum i Wszystko, która zostaje udzielona grupie myszy. Jak twierdzi Adams, myszy są trójwymiarowymi postaciami hiperinteligentnych istot wielowymiarowych, które zbudowały ogromny superkomputer, zdolny udzielić odpowiedzi na wspomniane Pytanie. Po siedmiu i pół milionach lat uzyskują ją: jest to właśnie czterdzieści dwa.
Za chwilę jednak komputer stwierdził, że tak naprawdę nie wiedział do końca, jakie pytanie zostało mu zadane. Pod koniec jednego z tomów serii dowiadujemy się jednak, cóż to było za pytanie:

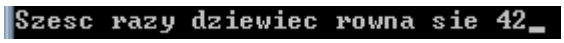
> Co otrzymamy, jeżeli pomnożymy sześć przez dziewięć?

Odpowiedź: czterdzieści dwa. Brzmi to zupełnie nonsensownie, zważywszy że 6×9 to przecież 54. A jednak to prawda - aby się o tym przekonać, popatrz na poniższy program:

```
// FortyTwo - odpowiedź na najważniejsze pytanie Wszechświata

#include <iostream>
#include <conio.h>

#define SZESC        1 + 5
#define DZIEWIEC     8 + 1
```

```
int main()
{
    std::cout << "Szesc razy dziewiec rowna sie " << SZESC * DZIEWIEC;
    getch();

    return 0;
}
```

Jak można zobaczyć, rzeczywiście drukuje on liczbę 42:



Screen 44. Komputer prawdę ci powie...

Czyżby więc była to faktycznie tak magiczna liczba, iż specjalnie dla niej naginane są zasady matematyki?... Niestety, wyjaśnienie jest bardziej prozaiczne. Spójrzmy tylko na wyrażenie `SZESC * DZIEWIEC`. Jest ono rozwijane do postaci:

```
1 + 5 * 8 + 1
```

Tutaj zaś, zgodnie z ważnymi od początku do końca Wszechświata regułami arytmetyki, pierwszym obliczanym działaniem jest mnożenie. Ostatecznie więc mamy `1 + 40 + 1`, czyli istotnie `42`.

Nie musimy jednak wierzyć temu prostego wytłumaczeniu. Czyż nie lepiej sądzić, że nasz poczciwy preprocesor ma dostęp do rozwiązań niewyjaśnionych od wieków zagadek Uniwersum?...

Predefiniowane makra kompilatora

Istnieje kilka makr, których definiowaniem zajmuje się sam kompilator. Dostarczają one kilku użytecznych informacji związanych z nim samym oraz z przebiegiem kompilacji. Dane te mogą być często przydatne przy usuwaniu błędów, więc przyjrzyjmy się im.

We wszystkich poniższych nazwach makr długie kreski oznaczają dwa znaki podkreślenia. Tak więc __ oznacza dwukrotne wpisanie znaku _, a nie jedną długą kreskę.

Numer linii i nazwa pliku

Jednymi z najbardziej przydatnych makr są `__FILE__` i `__LINE__`. Pozwalają one na wykrycie miejsca w kodzie, gdzie np. zaszedł błąd wpływający na działanie programu.

Numer wiersza

Makro `__LINE__` zostaje przez preprocesor zamienione na numer wiersza w aktualnie przetwarzanym pliku źródłowym. Wiersze liczą się od 1 i obejmują także dyrektywy oraz puste linijki. Zatem w poniższym programie:

```
#include <iostream>
#include <conio.h>

int main()
{
    std::cout << "Wypisanie tekstu w wierszu " << __LINE__ << std::endl;
    return 0;
}
```

liczbą pokazaną na ekranie będzie 6. Można też zauważyć, że sam kompilator posługuje się tą nazwą, gdy pokazuje nam komunikat o błędzie podczas nieudanej kompilacji programu.

Nazwa pliku z kodem

Do pary z numerem wiersza potrzebujemy jeszcze nazwy pliku, aby precyzyjnie zlokalizować błąd. Tę zaś zwraca makro `__FILE__`:

```
std::cout << "Ten kod pochodzi z modułu " << __FILE__;
```

Jest ono zamieniane na nazwę pliku kodu, ujętą w podwójne cudzysłowy - właściwe dla napisów w C++. Zatem jeśli nasz moduł nazywa się *main.cpp*, to `__FILE__` zostanie zastąpione przez `"main.cpp"`.

Dyrektywa `#line`

Informacje podawane przez `__LINE__` i `__FILE__` możemy zmienić, umieszczając te makra w innych miejscach (plikach?). Ale możliwe jest też oszukanie preprocesora za pomocą dyrektywy `#line`:

```
#line wiersz ["plik"]
```

Gdy z niej skorzystamy, to preprocesor uzna, że umieszczona ona została w linijce o numerze *`wiersz`*. Jeżeli podamy też nazwę pliku, to wtedy także oryginalna nazwa modułu zostanie unieważniona przez tę podaną. Oczywiście nie fizycznie: sam plik pozostanie nietknięty, a tylko preprocesor będzie myślał, że zajmuje się innym plikiem niż w rzeczywistości.

Osobiście nie sądzę, aby świadome oszukiwanie miało tu jakiś głębszy sens. (Nad)używając dyrektywy `#line` możemy łatwo stracić orientację nawet w programie, który obficie drukuje informacje o sprawiających problemy miejscach w kodzie.

Data i czas

Innym rodzajem informacji, jakie można wkompilować do wynikowego programu, jest data i czas jego zbudowania, ewentualnie modyfikacji kodu. Służą do tego dyrektywy `__DATE__`, `__TIME__` oraz `__TIMESTAMP__`.

Zwróćmy jeszcze uwagę, że polecenia te absolutnie **nie służą do pobierania bieżącego czasu** systemowego. Są one tylko zamieniane na dosłowne stałe, które w niezmienionej postaci są przechowywane w gotowym programie i np. wyświetlane wraz z informacją o wersji.
Natomiast do uzyskania aktualnego czasu używamy znanych funkcji `time()`, `localtime()`, itp. z pliku nagłówkowego *ctime*.

Czas kompilacji

Chcąc zachować w programie datę i godzinę jego kompilacji, stosujemy dyrektywy - odpowiednio: `__DATE__` oraz `__TIME__`. Preprocesor zamienia je na datę w formacie *Mmm dd yy* i na czas w formacie *hh:mm:ss*. Obie te wartości są literałami znakowymi, a więc ujęte w cudzysłowy.

Przykładowo, gdybym w chwili pisania tych słów skompilował poniższą linijkę kodu:

```
std::cout << "Kompilacja wykonana w dniu " << __DATE__ <<
          << " o godzinie " << __TIME__ << std::endl;
```

to w programie zapisana zostałaby data "Jul 14 2004" i czas "18:30:51". Uruchamiając program za minutę, pół godziny czy za dziesięć lat ujrzałbym tę samą datę i ten sam czas, ponieważ byłyby one **wpisane na stałe** w pliku EXE.
Z tego powodu data i czas kompilacji mogą być użyte jako prymitywny sposób podawania wersji programu.

Czas modyfikacji pliku

Makro __TIMESTAMP__ jest nieco inne. Nie podaje ono czasu kompilacji, lecz datę i czas ostatniej modyfikacji pliku z kodem. Jest to dana w formacie *Ddd Mmm d hh:mm:ss yyyy*, gdzie *Ddd* jest skrótem dnia tygodnia, zaś *d* jest numerem dnia miesiąca.

Popatrz na przykład. Jeśli wpiszę teraz do modułu poniższą linijkę i zachowam plik kodu:

```
std::cout << "Data ostatniej modyfikacji " << _TIMESTAMP__;
```

to w programie zapisany zostanie napis "Wed Jul 14 18:38:37 2004". Będzie tak niezależnie od chwili, w której skompiluję program - chyba że do czasu jego zbudowania poczynię w kodzie jeszcze jakieś poprawki. Wówczas __TIMESTAMP__ zmieni się odpowiednio, wyświetlając moment zapisywania ostatnich zmian.

Piszę tu, iż __TIMESTAMP__ coś wyświetli, ale to oczywiście skrót myślowy. Naprawdę to makro zostanie zastąpione przez preprocesor odpowiednim napisaem, zaś jego prezentacją zajmie się rzecz jasna strumień wyjścia.

Typ kompilatora

Jest jeszcze jedno makro, zdefiniowane zawsze w kompilatorach języka C++. To __cplusplus. Nie ma ono żadnej wartości, gdyż liczy się sama jego obecność. Pozwala ona na wykorzystanie tzw. kompilacji warunkowej, którą poznamy za jakiś czas, do rozróżniania kodu w C i w C++.
Dla nas, nieużywających wcześniej języka C, makro to nie jest więc zbyt praktycze, ale w czasie migracji starszego kodu do nowego języka okazywało się bardzo przydatne. Poza tym wiele kompilatorów C++ potrafi udawać kompilatory jego poprzednika w celu budowania wykonywalnych wersji starych aplikacji. Jeśli włączylibyśmy taką opcję w naszym ulubionym kompilatorze, wtedy makro __cplusplus nie byłoby definiowane przed rozpoczęciem pracy preprocesora.

Inne nazwy

Powyższe nazwy są zdefiniowane w każdym kompilatorze choć trochę zgodnym ze standardem C++. Wiele z nich definiuje jeszcze inne: przykładowo, Visual C++ udostępnia makra __FUNCTION__ i __FUNCSIG__, które wewnątrz bloków funkcji są zmieniane w ich nazwy i sygnatury (nagłówki).
Ponadto, kompilatory pracujące w środowisku Windows definiują też nazwy w rodzaju _WIN32 czy _WIN64, pozwalające określić „bitowość" platform tego systemu.

Po inne predefiniowane makra preprocesora musisz zajrzeć do dokumentacji swojego kompilatora. Jeśli używasz Visual C++, to będzie nią oczywiście MSDN.

Makra parametryzowane

Bardziej zaawansowany rodzaj makr to **makra parametryzowane**, czyli **makrodefinicje**. Z wyglądu przypomniają one nieco funkcje, choć funkcjami nie są. To po prostu nieco bardziej wyrafinowe polecenia na preprocesora, instruujące go, jak powinien zamieniać jeden tekst kodu w inny.

Nie wydaje się to szczególnie skomplikowane, jednak wokół makrodefinicji narosło mnóstwo mitów i fałszywych stereotypów. Chyba żaden inny element języka C++ nie wzbudza tylu kontrowersji co do jego prawidłowego użycia, a wśród nich przeważają opinie bardzo skrajne. Mówią one, ze makrodefinicje są całkowicie przestarzałe i nie powinny być w ogóle stosowane, gdyż z powodzeniem zastępują je inne elementy języka. Jak każde radykalne sądy, nie są to zdania słuszne. To prawda jednak, że obecnie pole zastosowań makrodefinicji (i makr w ogóle) zawężyło się znacznie. Nie jest to aczkolwiek wystarczającym powodem, ażeby usprawiedliwiać nim nieznajomość tej ważnej części języka. Zobaczmy zatem, co jest przyczyną tego całego zamieszania.

Definiowanie parametrycznych makr

Makrodefinicje nazywamy parametryzowanymi makrami, ponieważ mają one coś w rodzaju parametrów. Nie są to jednak konstrukcje podobne do parametrów funkcji - w dalszej części sekcji przekonamy się, dlaczego.

Na razie spojrzyjmy na przykładową definicję:

```
#define SQR(x)        ((x) * (x))
```

W ten sposób zdefiniowaliśmy makro `SQR()`, posiadające jeden parametr - nazwaliśmy go tu `x`. Treścią makra jest natomiast wyrażenie `((x) * (x))`. Jak ono działa?
Otóż, jeśli preprocesor napotka w programie na „wywołanie”:

```
SQR(cokolwiek)
```

to zamieni je na wyrażenie:

```
((cokolwiek) * (cokolwiek))
```

Tym *`cokolwiek`* może być teoretycznie dowolny tekst (przypominam do znudzenia, że preprocesor operuje na tekście programu), ale sensowne jest tam wyłącznie podanie wartości liczbowej[100]. Wszelkie eksperymentowanie np. z łańcuchami znaków skończy się komunikatem o błędzie składniowym albo niedozwolonym użyciu operatora `*`.

Powiedzmy jeszcze, dlaczego słowo ‘wywołanie’ wziąłem w cudzysłów, choć pewnie domyślasz się tego. Tak, **makro nie jest żadną funkcją**, więc jego użycie nie oznacza przejścia do innej części programu. Makrodefinicja jest tylko poleceniem na preprocesora, mówiącym mu, w jaki sposób zmienić to wywołaniopodobne wyrażenie `SQR(x)` na inny fragment kodu, wykorzystujący symbol `x`. W tym przypadku jest to iloczyn dwóch „zmiennych” `x`, czyli kwadrat podanego wyrażenia.

A jak wygląda to makro w akcji? Bardzo prosto:

```
int nLiczba;

std::cout << "Podaj liczbę: ";
std::cin >> nLiczba;
std::cout << "Kwadrat liczby " << nLiczba << " to " << SQR(nLiczba);
```

Użycie makra w postaci `SQR(nLiczba)` zostanie tu zamienione na `((nLiczba) * (nLiczba))`, zatem w wyniku rzeczywiście dostaniemy kwadrat podanej liczby.

[100] Lub ogólnie: każdego typu danych, dla którego zdefiniowaliśmy (lub zdefiniował kompilator) działanie operatora `*`. O (prze)definiowaniu znaczeń operatorów mówi następny rozdział.

Kilka przykładów

Dla utrwalenia przyjrzyjmy się jeszcze innym przykładom makrodefinicje.

Wzory matematyczne

Proste podniesienie do kwadratu to nie jedyne działanie, jakie możemy wykonać poprzez makro. Prawie każdy prosty wzór daje się zapisać w postaci odpowiedniej makrodefinicji - spójrzmy:

```
#define CB(x)        ((x) * (x) * (x))
#define SUM_1_n(n)   ((n) * ((n) + 1) / 2)
#define POLE(a)      SQR(a)
#define POLE(a,b)    ((a) * (b))
```

Możemy tu zauważyć kilka faktów na temat parametryzowanych makr:

- mogą one korzystać z już zdefiniowanych makr (parametryzowanych lub nie) oraz wszelkich innych informacji dostępnych w czasie kompilacji - jak choćby obiektów `const`
- możliwe jest zdefiniowanie makra z więcej niż jednym parametrem. Wtedy jednak dla bezpieczeństwa lepiej **nie stawiać spacji po przecinku**, gdyż niektóre kompilatory uznają **każdy biały znak** za koniec nazwy i rozpoczęcie treści makra. W nazwach typu `POLE(a,b)` i podobnych nie wpisujmy więc żadnych białych znaków
- makrodefinicje można „przeciążać", tj. zdefiniować kilka sztuk o tej samej nazwie. Ponieważ jednak parametry makr nie mają przypisanych typów, poszczególne wersje makr o identycznych nazwach muszą się różnić liczbą argumentów

Jeśli chodzi o łatwo zauważalne, intensywne użycie nawiasów w powyższych definicjach, to wyjaśni się ono za parę chwil. Sądzę jednak, że pamiętając o doświadczeniach z makrami-stałymi, domyślasz się ich roli…

Skracanie zapisu

Podobnie jak makra bez parametrów, makrodefinicje mogą przydać się do skracania często używanych fragmentów kodu. Oferują one jeszcze możliwość ogólnego zdefiniowania takiego fragmentu, bez wyraźnego podania niektórych nazw np. zmiennych, które mogą się zmieniać w zależności od miejsca użycia makra.

A oto potencjalnie użyteczny przykład:

```
#define DELETE(p)   { delete (p); (p) = NULL; }
```

Makro `DELETE()` jest przeznaczone do usuwania obiektu, na który wskazuje wskaźnik `p`. Dodatkowo jeszcze dokonuje ono zerowania wskaźnika - dzięki temu będzie można uchronić się przed omyłkowym odwołaniem do zniszczonego obiektu. Zerowy wskaźnik można bowiem łatwo wykryć za pomocą odpowiedniego warunku `if`.

Jeszcze jeden przykład:

```
#define CLAMP(x, a, b)      { if ((x) <= (a)) (x) = (a);
                              if ((x) >= (b)) (x) = (b); }
```

To makro pozwala z kolei upewnić się, że zmienna (liczbowa) podstawiona za `x` będzie zawierać się w przedziale `<a; b>`. Jego normalne użycie w formie:

```
CLAMP(nZmienna, 1, 10)
```

zostanie rozwinięte do kodu:

```
{ if ((nZmienna) <= (1)) (nZmienna) = (1);
  if ((nZmienna) >= (10)) (nZmienna) = (10); }
```

po wykonaniu którego będziemy pewni, że `nZmienna` zawiera wartość równą co najmniej `1` i co najwyżej `10`.

Przypominam o nawiasach klamrowych w definicjach makr. Jak sądzę pamiętasz, że chronią one przed nieprawidłową interpretacją kodu makra w jednolinijkowych instrukcjach `if` oraz pętlach.

Operatory preprocesora

W definicjach makr możemy korzystać z kilku operatorów, niedozwolonych nigdzie indziej. To specjalne operatory preprocesora, które za chwilę zobaczymy przy pracy.

Sklejacz

Sklejacz (ang. *token-pasting operator*) jest też często nazywany **operatorem łączenia** (ang. *merging operator*). Obie nazwy są adekwatne do działania, jakie ten operator wykonuje. W kodzie makr jest on reprezentowany przez dwa znaki płotka (*hash*) - `##`.

Sklejacz łączy ze sobą dwa identyfikatory, czyli nazwy, w jeden nowy identyfikator. Najlepiej prześledzić to działanie na przykładzie:

```
#define FOO          foo##bar
```

Wystąpienie `FOO` w programie zostanie przez preprocesor zamienione na złączenie nazw `foo` i `bar`. Będzie to więc `foobar`.

Operator łączący przydaje się też w makrodefinicjach, ponieważ potrafi działać na ich argumentach. Spójrzmy na takie oto przydatne makro:

```
#define UNICODE(text)     L##text
```

Jego „wywołanie” z jakąkolwiek dosłowną stałą napisową spowoduje jej interpretację jako łańcuch znaków Unicode. Przykładowo:

```
UNICODE("Wlazł kotek na płotek i spadł")
```

zmieni się na:

```
L"Wlazł kotek na płotek i spadł"
```

czyli napis zostanie zinterpretowany jako składający się z 16-bitowych, „szerokich” znaków.

Operator łańcuchujący

Drugim z operatorów preprocesora jest **operator łańcuchujący** (ang. *stringizing operator*). Symbolizuje go jeden znak płotka (*hash*) - `#`, zaś działanie polega na ujęciu w podówjne cudzysłowy (`""`) nazwy, którą owym płotkiem poprzedzimy.
Popatrzmy na takie makro:

```
#define STR(string)        #string
```

Działa ono w prosty sposób. Jeśli podamy mu jakąkolwiek nazwę czegokolwiek, np. tak:

```
STR(jakas_zmienna)
```

to w wyniku rozwinięcia zostanie ona zastąpiona przez napis ujęty w cudzysłowy:

```
"jakas_zmienna"
```

Podana nazwa może składać z kilku wyrazów - także zawierających znaki specjalne, jak cudzysłów czy ukośnik:

```
STR("To jest tekst w cudzyslowach")
```

Zostaną one wtedy zastąpione odpowiednimi sekwencjami ucieczki, tak że powyższy tekst zostanie zakodowany w programie w sposób dosłowny:

```
"\"To jest tekst w cudzyslowach\""
```

W programie wynikowym zobaczylibyśmy więc napis:

```
"To jest tekst w cudzysłowach"
```

Byłby on więc identycznie taki sam, jak argument makra `STR()`.

Visual C++ posiada jeszcze **operator znakujący** (ang. *charazing operator*), któremu odpowiada symbol `#@`. Operator ten powoduje ujęcie podanej nazwy w apostrofy.

Niebezpieczeństwa makr

Niechęć wielu programistów do używania makr nie jest bezpodstawna. Te konstrukcje językowe kryją w sobie bowiem kilka pułapek, których umiejscowienie należy znać. Dzięki temu można je omijać - same te pułapki, albo nawet makra w całości. Zobaczmy więc, na co trzeba zwrócić uwagę przy korzystaniu z makrodefinicji.

Brak kontroli typów

Początek definicji sparametryzowanego makra (zaraz za `#define`) przypomina deklaracją funkcji, lecz bez określenia typów. Nie podajemy tu zarówno typów parametrów, jak i typów „zwracanej wartości”. Dla preprocesora wszystko jest bowiem zwyczajnym tekstem, który ma być jedynie przetransformowany według podanego wzoru.

Potencjalnie więc może to rodzić problemy. Na szczęście jednak są one zawsze wykrywane już ne etapie kompilacji. Jest tak, gdyż o ile preprocesor posłusznie rozwninie wyrażenie typu:

```
SQR("Tekst")
```

do postaci:

```
(("Tekst") * ("Tekst"))
```

o tyle kompilator nigdy nie pozwoli na mnożenie dwóch napisów. Taka operacja jest przecież kompletnie bez sensu.

Dezorientację może jedynie wzbudzać komunikat o błędzie, jaki dostaniemy w tym przypadku. Nie będzie to coś w rodzaju: "Błędny argument makra", bo dla kompilatora makra już tam nie ma - jest tylko iloczyn dwóch łańcuchów. Błąd będzie więc dotyczył niewłaściwego użycia operatora `*`, co nie od razu może nasuwać skojarzenia z makrami.

Jeśli więc kompilator zgłasza nam dziwnie wyglądający błąd na (z pozoru) niewinnej linijce kodu, to sprawdźmy przede wszystkim, czy nie ma w niej niewłaściwego użycia makrodefinicji.

Parokrotne obliczanie argumentów

Błędy związane z typami wyrażeń nie są zbyt kłopotliwe, gdyż wykrywane są już w trakcie kompilacji. Inne problemy z makrami nie są aż tak przyjemne…

Rozpatrzmy teraz taki kod:

```
int nZmienna = 7;
std::cout << SQR(nZmienna++) << std::endl;
std::cout << nZmienna;
```

Kompilator z pewnością nie będzie miał nic przeciwko niemu, ale jego działanie może być co najmniej zaskakujące. Wedle wszelkich przewidywań powinien on przecież wydrukować liczby 49 i 8, prawda?…
Dlaczego więc wynik jego wykonania przedstawia się tak:

```
56
9
```

Aby dociec rozwiązania, rozpiszmy druga linijkę tak, jak robi to preprocesor:

```
std::cout << ((nZmienna++) * (nZmienna++)) << std::endl;
```

Widać wyraźnie, że `nZmienna` jest tu inkrementowana dwukrotnie. Pierwsza postinkrementacja zwraca wprawdzie wyniku 7, ale po niej `nZmienna` ma już wartość 8, zatem druga inkrementacja zwróci w wyniku właśnie 8. Obliczymy więc iloczyn 7×8, czyli 56.
Ale to nie wszystko. Druga inkrementacja zwiększy jeszcze wartość 8 o jeden, zatem `nZmienna` będzie miała ostatecznie wartość 9. Obie te niespodziewane liczby ujrzymy na wyjściu programu.

Jaki z tego wniosek? Ano taki, że wyrażenia podane jako argumenty makr są obliczane tyle razy, ile razy występują w ich definicjach. Przyznasz, że to co najmniej nieoczekiwane zachowanie…

Priorytety operatorów

Pora na akt trzeci dramatu. Obiecałem wcześniej, że wyjaśnię, dlaczego tak gęsto stawiam nawiasy w definicjach makr. Jeśli uważnie czytałeś sekcję o makrach-stałych, to najprawdopodobniej już się tego domyślasz. Wytłumaczmy to jednak wyraźnie.

Najlepiej będzie przekonać o roli nawiasów na przykładzie, w którym ich nie ma:

```
#define SUMA(a,b,c)           a + b + c
```

Użyjemy teraz makra `SUMA()` w takim oto kodzie:

```
std::cout << 4 * SUMA(1, 2, 3);
```

Jaką liczbę wydrukuje nam program? Oczywiście 24… Zaraz, czy aby na pewno? Kompilacja i uruchomienie kończy się przecież rezultatem:

```
9
```

Co się zatem stało? Ponownie winne jest wyrażenie wykorzystujące makra. Preprocesor rozwinie je przecież do postaci:

```
4 * 1 + 2 + 3
```

co wedle wszelkich prawideł rachunku na liczbach (i pierwszeństwa operatorów w C++) każe najpierw wykonać mnożenie `4 * 1`, a dopiero potem resztę dodawania. Wynik jest więc zupełnie nieoczekiwany.

Jak się też zdążyliśmy wcześniej przekonać, podobną rolę jak nawiasy okrągłe w makrach-wyrażeniach pełnią nawiasy klamrowe w makrach zastępujących całe instrukcje.

Zalety makrodefinicji

Z lektury poprzedniego paragrafu wynika więc, że stosowanie makrodefinicji wymaga ostrożności zarówno w ich definiowaniu (nawiasy!), jak i późniejszych użyciu (przekazywanie prostych wyrażeń). Co zaś zyskujemy w zamian, jeśli zdecydujemy na stosowanie makr?

Efektywność

Na każdym kroku wyraźnie podkreślam, jak działają makrodefinicje. To nie są funkcje, które program wywołuje, lecz dosłowny kod, który zostanie wstawiony w miejsce użycia przez preprocesor.

Co z tego wynika? Otóż z pozoru jest to bardzo wyraźna zaleta. Brak konieczności skoku w inne miejsce programu - do funkcji - oznacza, że nie trzeba wykonywać wszelkich czynności z tym związanych.
Nie trzeba zatem angażować pamięci stosu, by zachować aktualny punkt wykonania oraz przekazać parametry. Nie trzeba też szukać w pamięci operacyjnej miejsca, gdzie rezyduje funkcja i przeskakiwać do niego. Wreszcie, po skończonym wykonaniu funkcji nie trzeba zdejmować ze stosu adresu powrotnego i przy jego pomocy wracać do miejsca wywołania.

Funkcje inline

A jednak te zalety nie są wcale argumentem przeważającym na korzyść makr. Wszystko dlatego, że C++ umożliwia skorzystanie z nich także w odniesieniu do zwykłych funkcji. Tworzymy w ten sposób **funkcje rozwijane w miejscu wywołania** - albo krótko: **funkcje *inline***.

Są tą funkcje pełną gębą i dlatego zupełnie nie dotyczą ich problemy związane z wielokrotnym obliczaniem wartości parametrów czy priorytetami operatorów. Działają one po prostu tak, jakbyśmy się tego spodziewali po normalnych funkcjach, a ponadto posiadają też zalety makrodefinicji. Funkcje *inline* nie są więc faktycznie wywoływane podczas działania programu, lecz ich kod zostaje wstawiony (rozwinięty) w miejscu wywołania podczas kompilacji programu. Dzieje się to zupełnie bez ingerencji programisty w sposób wywoływania funkcji.
Jedyne, co musi on zrobić, to poinformować kompilator, które funkcje mają być rozwijane. Czyni to, **przenosząc ich definicje do pliku nagłówkowego** (to ważne![101]) i opatrując przydomkiem `inline`, np.:

```
inline int Sqr(int a)      { return a * a; }
```

[101] Jest tak, gdyż pełna definicja funkcji *inline* (a nie tylko prototyp) musi być znana w miejscu wywołania funkcji - tak, aby jej treść mogła być wstawiona bezpośrednio do kodu w tym miejscu.

„Wspaniale!”, możesz krzyknąć, „Odtąd wszystkie funkcje będę deklarował jako *inline*!” Chwileczkę, nie tędy droga. Musisz być świadom, że wstawianie kodu dużych funkcji w miejsce każdego ich wywołania powodowałoby rozdęcie kodu do sporych rozmiarów. Duży rozmiar mógłby nawet spowolnić wykonanie programu, zajmującego nadzwyczajnie dużo miejscu w pamięci operacyjnej. Na funkcjach *inline* można się więc poślizgnąć. Lepiej zatem nie opatrywać modyfikatorem `inline` żadnych funkcji, które mają więcej niż kilka linijek. Na pewno też nie powinny to być funkcje zawierające w swym ciele pętle czy inne rozbudowane konstrukcje językowe (typu `switch` lub wielopoziomowych instrukcji `if`).

Miło jest jednak wiedzieć, że obecne kompilatory są po naszej stronie, jesli chodzi o funkcje *inline*. Dobry kompilator potrafi bowiem zrobić analizę zysków i strat z zastosowania `inline` do konkretnej funkcji: jeśli stwierdzi, że w danym przypadku rozwijanie urągałoby szybkości programu, nie przeprowadzi go. Dla prostych funkcji (dla których `inline` ma największy sens) kompilatory zawsze jednak ulegają naszym żądaniom.

W Visual C++ jest dodatkowe słowo kluczowe `__forceinline`. Jego użycie zamiast `inline` sprawia, że kompilator na pewno rozwinie daną funkcję w miejscu wywołania, ignorując ewentualne uszczerbki na wydajności. VC++ ma też kilka dyrektyw `#pragma`, które kontrolują rozwijanie funkcji *inline* - możesz o nich przeczytać w dokumentacji MSDN.

Warto też wiedzieć, że metody klas definiowane wewnątrz bloków `class` (lub `struct` i `union`) są **automatycznie *inline***. Nie musimy opatrywać ich żadnym przydomkiem. Jest to szczególnie korzystne dla metod dostępowych do pól klasy.

Makra kontra funkcje inline

Cóż więc wynika z zapoznania się z funkcjami *inline*? Ano to, że powinniśmy je stosować zawsze wtedy, gdy przyjdzie nam ochota na wykorzystanie makrodefinicji. Funkcje *inline* są po prostu lepsze, gdyż łączą w sobie zarówno zalety zwykłych funkcji, jak i zalety makr.

Brak kontroli typów

Wydawałoby się jednak, że jest jedna sytuacja, gdy makra mają przewagę nad zwykłymi funkcjami. Ta wyższość ujawnia się w cesze, którą poprzednio wskazaliśmy jako ich słabość: w braku kontroli typów.

Otóż często jest to wręcz pożądana właściwość. Nie wiem czy zauważyłeś, ale większość zdefiniowanych przez nas makr działa równie dobrze dla liczb całkowitych, jak i rzeczywistych. Działa dla każdego typu zmiennych liczbowych:

```
SQR(-14)        // int
SQR(12u)        // unsigned
SQR(3.14f)      // float
SQR(-87.56)     // double
```

Łatwo to wyjaśnić. Preprocesor zamieni po prostu każde użycie makra na odpowiedni iloczyn, zapisany w sposób dosłowny w kodzie wysłanym do kompilatora. Ten zaś potraktuje te wyrażenia jak każde inne.

Gdybyśmy chcieli podobny efekt uzyskać przy pomocy funkcji *inline*, to zapewne pierwszym pomysłem byłoby napisanie kilku(nastu?) przeciążonych wersji funkcji. To jednak nie jest konieczne: C++ potrafi bowiem stosować w kontekście normalnych funkcji także i tę cechę makra, jaką jest niezależność od typu. Poznamy bowiem wkrótce mechanizm szablonów, który pozwala na takie właśnie zachowanie.

Ciekawostka: funkcje szablonowe

Niecierpliwym pokażę już teraz, w jaki sposób makro `SQR()` zastąpić funkcją szablonową. Odpowiedni kod może wyglądać tak:

```
template <typename T> inline T Sqr(T a)      { return a * a; }
```

Powyższy szablon funkcji (tak to się nazywa) może być stosowany dla każdego typu liczbowego, a nawet więcej - dla każdego typu obsługującego operator `*`. Posiada przy tym te same zalety co zwykłe funkcje i funkcje *inline*, a pozbawiony jest typowych dla makr kłopotów z wielokrotnym obliczaniem argumentów i nawiasami.
W jednym z przyszłych rozdziałów poznamy dokładnie mechanizm szablonów w C++, który pozwala robić tak wspaniałe rzeczy bardzo małym kosztem.

Zastosowania makr

Czytelnicy chcący znaleźć uzasadnienie dla wykorzystania makr, mogą się poczuć zawiedzeni. Wyliczyłem bowiem wiele ich wad, a wszystkie zalety okazywały się w końcu zaletami pozornymi. Takie wrażenie jest w dużej części prawdziwe, lecz nie znaczy to, że makrach należy całkiem zapomnieć. Przeciwnie, należy tylko wiedzieć, gdzie, kiedy i jak z nich korzystać.

Nie korzystajmy z makr, lecz z obiektów `const`

Przede wszystkim nie powinniśmy używać makr tam, gdzie lepiej sprawdzają się inne konstrukcje języka. Jeżeli kompilator dostarcza nam narzędzi zastępujących dane pole zastosowań makr, to zawsze będzie to lepszy mechanizm niż same makra.

Dotyczy to na przykład stałych. Już na samym początku kursu podkreśliłem, żeby stosować przydomek `const` do ich definiowania. Użycie `#define` pozbawia bowiem stałe cennych cech „niezmiennych zmiennych” - typu, zasięgu oraz miejsca w pamięci.

Nie korzystajmy z makr, lecz z (szablonowych) funkcji inline

Podobnie nie powinniśmy korzystać z makrodefinicji, by zyskać na szybkości programu. Te same efekty szybkościowe osiągniemy bowiem za pomocą funkcji *inline*, zaś przy okazji nie pozbawimy się wygody i bezpieczeństwa, jakie daje ich stosowanie (w przeciwieństwie do makr).

A jeśli chodzi o niewrażliwość na typy danych, to obecnie może to być dla ciebie zaletą. Kiedy jednak poznasz technikę szablonów, także i ten argument straci swoją ważność.

Korzystajmy z makr, by zastępować powtarzające się fragmenty kodu

Jak więc poprawnie stosować makra? Najważniejsze jest, aby zapamiętać, czym one są. Powiedzieliśmy sobie dotąd, czym makra nie są - nie są stałymi i nie są funkcjami. Makra to najsampierw sposób na zastąpienie jednego fragmentu kodu innym. Używamy ich więc wtedy, gdy zauważymy czsto powtarzające się sekwencje dwóch-trzech instrukcji, których wyodrębnienie w osobnej funkcji nie jest możliwe, lecz których ręczne wpisywanie staje się nużące. Dla takich właśnie sytuacji stworzono makra.

Korzystajmy z makr, by skracać sobie zapis

Makra są narzędzami do operacji na tekście - tekście programu, czyli kodzie. Stosujmy je więc, aby dokonywać takich automatycznych działań.
Jeden przykład takiego zastosowania już podałem: to bezpieczne zniszczenie obiektu połaczone z wyzerowaniem wskaźnika. Innym może być chociażby pobranie liczby elementów niedynamicznej tablicy:

```
#define ELEMENTS(tab)      ((sizeof(tab) / sizeof((tab)[0]))
```

Znanych jest wiele podobnych i przydatnych sztuczek, szczególnie z wykorzystanie operatorów preprocesora - `#` i `##`. Być może niektóre z nich sam odkryjesz lub znajdziesz w innych źródłach.

Korzystajmy z makr zgodnie z ich przeznaczeniem

Na koniec nie mogę jeszcze nie wspomnieć o bardzo ważnym zastosowaniu makr, przewidzianym przez twórców języka. Zastosowanie to przetrwało próbę czasu i nikt nawet myśli o jego zastąpieniu czy likwidacji.
Tym polem wykorzystania makr jest kompilacja warunkowa. Ten użyteczny sposób na kontrolę procesu kompilacji programu jest tematem następnego podrozdziału.

Kontrola procesu kompilacji

Preprocesor wkracza do akcji, przeglądając kod jeszcze zanim zrobi to kompilator. Sprawia to, że możliwe jest wykorzystanie go do sprawowania kontroli nad procesem kompilacji programu. Możemy określić, jakie jego fragmenty mają pojawić się w wynikowym pliku EXE, a jakie nie. Podejmowanie takich decyzji nazywamy **kompilacją warunkową** (ang. *conditional compilation*).

Do czego może to się przydać? Przede wszystkim pozwala to dołączyć do programu dodatkowy kod, pomocny w usuwaniu z niego błędów. Zazwyczaj jest to kod wyświetlający pewne pośrednie wyniki obliczeń, logujący przebieg pewnych czynności lub prezentujący co określony czas wartości kluczowych zmiennych. Po zakończeniu testowania aplikacji możnaby było ów kod usunąć, ale jest przecież niewykluczone, że stanie się on przydatny w pracy nad kolejną wersją.
Wyjściem byłoby więc jego czasowe wyłączenie w momencie finalnego kompilowania. Najprostszym rozwiązaniem wydaje się użycie komentarza blokowego i jest to dobre wyjście - pod jednym warunkiem: że nasz kompilator pozwala na zagnieżdżanie takich komentarzy. Nie jest to wcale obowiązkowy wymóg i dlatego nie zawsze to się sprawdza. Komentowanie ma jeszcze jedną wadę: komentarze trzeba za każdym razem dodawać lub usuwać ręcznie. Po kilku-kilkunastu-kilkudziesięciu powtórzeniach kompilacji staje się to prawdziwą udręką.
A przecież można sprytniej. Kompilacja warunkowa pozwala bowiem w prosty sposób włączać i wyłączać kompilowanie określonego kodu w zależności od stanu pewnych ustalonych warunków.

Mechanizm ten ma jeszcze jedną zaletę, związana z przenośnością programów. Daje się to najbardziej odczuć w aplikacjach rozprowadzanych wraz z kodem źródłowym czy nawet wyłącznie w postaci źródłowego (programach na licencjach Open Source i GNU GPL). Takie programy mogą być teoretycznie kompilowane na wszystkich systemach operacyjnych i platformach sprzętowych, dla których istnieją kompilatory C++. W praktyce zależy to od warunkow zewnętrznych: wiadomo na przykład doskonale, że program dla środowiska Windows nie uruchomi się ani nie skompiluje w systemie Linux. Jednak nawet pomiędzy komputerami pracującymi pod kontrolą tych samych systemów operacyjnych występują różnice (zwłaszcza jeśli chodzi o Linux). Przykładowo, procesory tych komputerów mogą różnić się architekturą: obecnie dominują jednostki 32-bitowe, ale w wielu zastosowaniach mamy już procesory o 64 bitach w słowie maszynowym. Kompilatory wykorzystujące te procesory mają odmienną wielkość typu `int`: odpowiednio 4 i 8 bajtów. Może to rodzić problemy z zapisywaniem i odczytywaniem danych. Podobnych przykładów jest bardzo dużo, więc twórcy aplikacji rozprowadzanych jako kod muszą liczyć się z tym, że będą one kompilowane na bardzo różnych systemach. Technika kompilacji warunkowej pozwala przygotować się na wszystkie ewentualności.

Większość opisanych tu problemów dotyczy aczkolwiek systemów z wolnym kodem źródłowym, takich jak Linux. Stosowanie kontrolowanej kompilacji nie ogranicza się

jednak tylko do programów pracujących pod kontrolą takich systemów. Także wiele funkcji Windows jest dostępnych jedynie w określonych wersjach systemu, a chcąc z nich skorzystać musimy wprowadzić do kodu dodatkowe informacje. Zostaną one wykorzystane w kompilacji warunkowej.

Kontrolowanie kompilacji może więc dać dużo korzyści. Warto zatem zobaczyć, w jaki sposób to się odbywa.

Dyrektywy `#ifdef` i `#ifndef`

Wpływanie na proces kompilacji odbywa się za pomocą kilku specjalnych dyrektyw preprocesora. Teraz poznamy kilka pierwszych, między innymi tytułowe #ifdef i #ifndef. Najpierw jednak drobne przypomnienie makr.

Puste makra

Wprowadzając makra napomknąłem, że podawanie ich treści nie jest obowiązkowe. Mówiąc dosłownie, preprocesor uzna za całkowicie poprawną definicję:

```
#define MAKRO
```

Jeśli `MAKRO` wystąpi dalej w pliku kompilowanym, to zostanie po prostu usunięte. Nie będzie on zatem zbyt przydatne, jeśli chodzi o operacje na tekście programu. To jednak nie jest teraz istotne.
Ważne jest **samo zdefiniowanie** tego makra. Ponieważ zrobiliśmy to, preprocesor będzie wiedział, że taki symbol został mu podany i „zapamięta” go. Pozwala nam to na zastosowanie kompilacji warunkowej.

Przypomnijmy jeszcze, że możemy odwołać definicję makra dyrektywą `#undef`.

Dyrektywa `#ifdef`

Najprostszą i jedną z częściej używanych dyrektyw kompilacji warunkowej jest #ifdef:

```
#ifdef makro
   instrukcje
#endif
```

Jej nazwa to skrót od angielskiego *if defined*, czyli ‘jeśli zdefiniowane’. Dyrektywa `#ifdef` powoduje więc kompilacje kodu *`instrukcji`*, **jeśli zdefiniowane** jest *`makro`*. *`instrukcje`* mogą być wielolinijkowe; kończy je dyrektywa `#endif`.

`#ifdef` pozwala na czasowe wyłączenie lub włączenie określonego kodu. Typowym zastosowaniem tej dyrektywy jest pomoc w usuwaniu błędów, czyli debuggowaniu. Możemy objąć nią na przykład kod, który drukuje parametry przekazane do jakiejś funkcji:

```
void Funkcja(int nParametr1, int nParametr2, float fParametr3)
{
   #ifdef DEBUG
         std::cout << "Parametr 1: " << nParametr1 << std::endl;
         std::cout << "Parametr 2: " << nParametr2 << std::endl;
         std::cout << "Parametr 3: " << fParametr3 << std::endl;
   #endif

   // (kod funkcji)
}
```

Kod ten zostanie skompilowany tylko wtedy, jeśli wcześniej zdefiniujemy makro `DEBUG`:

```
#define DEBUG
```

Treść makra nie ma znaczenia, bo liczy się sam fakt jego zdefiniowania. Możemy więc pozostawić ją pustą. Po zakończeniu testowania usuniemy lub wykomentujemy tę definicję, a linijki drukujące parametry nie zostaną włączone do programu. Jeśli użyjemy `#ifdef` (lub innych dyrektyw warunkowych) większą liczbę razy, to oszczędzimy mnóstwo czasu, bo nie będziemy musieli przeszukiwać programu i oddzielnie komentować każdej porcji diagnostycznego kodu.

W wielu kompilatorach możemy wybrać tryb kompilacji, jak np. Debug (testowa) i Release (wydaniowa) w Visual C++. Różnią sie one stopniem optymalizacji i bezpieczeństwa, a także zdefiniowanymi makrami. W trybie Debug kompilator Microsoftu sam definiuje makro `_DEBUG`, którego obecność możemy testować.

Dyrektywa `#ifndef`

Przeciwnie do #ifdef działa druga dyrektywa - `#ifndef`:

```
#ifndef makro
   instrukcje
#endif
```

Ta opozycja polega na tym, że *`instrukcje`* ujęte w `#ifndef`/`#endif` zostaną skompilowane tylko wtedy, gdy *`makro`* nie jest zdefiniowane. `#ifndef` znaczy *if not defined*, czyli właśnie 'jeżeli nie zdefiniowane'.

Nawiązując do kolejnego przykładu, możemy użyć `#ifndef` w stosunku do kodu, który ma się kompilować wyłącznie w wersjach wydaniowych. Może to być choćby wyświetlanie ekranu powitalnego (ang. *splash screen*). Jego widok przy setnym, testowym uruchamianiu programu może być bowiem naprawdę denerwujący.

Dyrektywa `#else`

Do spółki z obiema dyrektywami `#ifdef` i `#ifndef` (a także z `#if`, opisaną w następnym paragrafie) wchodzi polecenie `#else`. Jak można się domyśleć, pozwala ono na wybór dwóch wariantów kodu: jednego, który jest kompilowany w razie zdefiniowania (`#ifdef`) lub niezdefiniowania (`#ifndef`) makra oraz drugiego - w przeciwnych sytuacjach:

```
#if[n]def makro
   instrukcje_1
#else
   instrukcje_2
#endif
```

Zastosowaniem dla tej dyrektywy może być na przykład system raportowania błędów. W trybie testowania można chcieć zrzutu całej pamięci programu, jeśli wystąpi w nim jakiś poważny błąd. W wersjach wydaniowych i tak nie możnaby było nic z krytycznym błędem zrobić, więc nie powinno się zmuszać (zdenerwowanego przecież) klienta do czekania na tak wyczerpującą operację. Wystarczy wtedy zapis wartości najważniejszych zmiennych.

Zwróćmy uwagę, że dyrektywa `#else` służy w tym przypadku wyłącznie naszej wygodzie. Równie dobrze poradzilibyście sobie bez niej, pisząc najpierw warunek z `#ifdef` (`#ifndef`), a potem z `#ifndef` (`#ifdef`).

Dyrektywa warunkowa #if

Uogólnieniem dyrektyw #ifdef i #ifndef jest dyrektywa #if:

```
#if warunek
   instrukcje
#endif
```

Przypomina ona instrukcję if, z tym że odnosi się do zagadnienia kompilowania lub niekompilowania wyszczególnionych instrukcji. #if ma też wersję z #else:

```
#if warunek
   instrukcje_1
#else
   instrukcje_2
#endif
```

Jak słusznie przypuszczasz, #if sprawi, że w przypadku spełnienia *warunku* skompilowany zostanie kod *instrukcje_1*, zaś w przeciwnym przypadku *instrukcje_2* (lub żaden, jeśli #else nie występuje).

Konstruowanie warunków

Co może być warunkiem? W ogólności wszystko, co znane jest preprocesorowi w momencie napotkania dyrektywy #if. Są to więc:

- wartości dosłownych stałych liczbowych, podane bezpośrednio w kodzie jako liczby, np. -8, 42 czy 0xFF
- wartości makr-stałych, zdefiniowane wcześniej dyrektywą #define
- wyrażenia z operatorem defined

A co z resztą stałych wartości, np. obiektami const?… Otóż one nie mogą (albo raczej nie powinny) być składnikami warunków #if. Jest tak, ponieważ obiekty te należą do kompilatora, a nie do preprocesora. Ten nie ma o nich pojęcia, gdyż zna tylko swoje makra #define. To jedyny przypadek, gdy mają one przewagę na stałymi const. Podobnie rzecz ma się z operatorem sizeof, który jest wprawdzie operatorem czasu kompilacji, ale **nie jest operatorem preprocesora**.

Gdyby #if rozpoznawało warunki z użyciem stałych const i operatora sizeof, nie mogłoby już być obsługiwane przez preprocesor. Musisz bowiem pamiętać, że dla preprocesora istnieją tylko jego dyrektywy, zaś cały tekst między nimi może być czymkolwiek (choć dla nas jest akurat kodem). Chcąc zmusić preprocesor do obsługi obiektów const i operatora sizeof należałoby w istocie obarczyć jego zadaniami kompilator.

Operator defined

Operator defined służy do sprawdzenia, czy dane makro zostało zdefiniowane. Warunek:

```
#if defined(makro)
```

jest więc równoważny z:

```
#ifdef makro
```

Natomiast dla #ifndef alternatywą jest:

```
#if !defined(makro)
```

Przewaga operatora defined na `#if[n]def` polega na tym, iż operator ten może występować w złożonych wyrażeniach, będących warunkami w dyrektywie `#if`.

Złożone warunki

`#if` jest podobna do `if` także pod tym względem, iż pozwala na stosowanie operatorów relacyjnych i logicznych w swoich warunkach. Nie zmienia to aczkolwiek faktu, że wszystkie argumenty tych operatorów muszą być znane w trakcie pracy preprocesora - a więc należeć do trzech grup, które podałem we wstępie do paragrafu.

Ta możliwość dyrektywy `#if` pozwala na warunkową kompilację kodu zależną od kilku warunków, na przykład:

```
#define MAJOR_VERSION      4
#define MINOR_VERSION      6

#if ((MAJOR_VERSION == 4) && (MINOR_VERSION >= 2))
     || (MAJOR_VERSION > 4))
   std::cout << "Ten kod skompiluje się tylko w wersji 4.2 lub nowszej";
#endif
```

Mogą w nich wystapić porównania makr-stałych, liczb wpisanych dosłownie oraz wyrażeń z operatorem `defined`. Wszystkie te części można natomiast łączyć znanymi operatorami logicznymi: `!`, `&&` i `||`.

Skomplikowane warunki kompilacji

To jeszcze nie wszystkie możliwości dyrektyw kompilacji warunkowej. Do bardziej wyszukanych należy ich zagnieżdżanie i spiętrzanie.

Zagnieżdżanie dyrektyw

Wewnątrz kodu zawartego między `#if[[n]def]` oraz `#else` i między `#else` i `#endif` mogą się znaleźć kolejne dyrektywy kompilacji warunkowej. Działa to w podobny sposób, jak zagnieżdżone instrukcje `if` w blokach kodu innych instrukcji `if`.
Spójrzmy na ten przykład[102]:

```
#define WINDOWS      1
#define WIN_NT       1

#define PLATFORM     WINDOWS
#define WIN_VER      WIN_NT

#if PLATFORM == WINDOWS
   #if WIN_VER == WIN_NT
         std::cout << "Program kompilowany na Windows z serii NT";
   #else
         std::cout << "Program kompilowany na Windows 9x lub ME";
   #endif
#else
   std::cout << "Nieznana platforma (DOS? Linux?)";
#endif
```

[102] To tylko przykład ilustrujący kompilację warunkową. Prawdziwa kontrola wersji systemu Windows, na której kompilujemy program, wymaga dołączenia pliku *windows.h* i kontrolowania makr o nieco innych nazwach i wartościach…

Jeśli zagnieżdżamy w sobie dyrektywy preprocesora, to stosujmy wcięcią podobne do instrukcji w normalnym kodzie. Nie wiedzieć czemu niektóre IDE (np. Visual C++) domyślnie wyrównują dyrektywy preprocesora w jednym pionie; wyłączmy im tę niepraktyczną opcję.
W Visual Studio .NET wybierzmy pozycję w menu *Tools*|*Options*, zaś w pojawiającym się oknie dialogowym przejdźmy do zakładki *Text Editor*|*C/C++*|*Tabs* i ustawmy opcję *Indenting* na *Block*.

Dyrektywa `#elif`

Czasem dwa warianty to za mało. Jeśli chcemy wybrać kilka możliwych dróg kompilacji, to należy zastosować dyrektywę `#elif`. Jej nazwa to skrót od *else if*, co mówi wszystko na temat roli tej dyrektywy.
Ponownie zerknijmy na przykładowy kod:

```
#define WINDOWS      1
#define LINUX        2
#define OS_2         3
#define QNX          4

#define PLATFORM     WINDOWS

#if PLATFORM == WINDOWS
   std::cout << "Kod kompilowany w systemie Windows";
#elif PLATFORM == LINUX
   std::cout << "Program budowany w systemie Linux";
#elif PLATFORM == OS_2
   std::cout << "Kompilacja na platformie systemu OS/2";
#elif PLATFORM == QNX
   std::cout << "Skompilowano w systemie QNX";
#endif
```

Do takich warunków pewnie znacznie lepsza byłaby dyrektywa typu `#switch`, lecz niestety preprocesor jest nie posiada.

Dyrektywa `#elif`, podobnie jak `#else`, może być także „doczepiona” do warunków `#ifdef` i `#ifndef`. Pamiętajmy jednak, że po niej musi nastąpić wyrażenie logiczne, a nie tylko nazwa makra.

Dyrektywa `#error`

Ostatnią z dyrektyw warunkowej kompilacji jest `#error`:

```
#error "komunikat"
```

Gdy preprocesor spotka ją na swojej drodze, wtedy jest to dla niego sygnałem, iż tok kompilacji schodzi na złe tory i powinien zostać przerwany. Czyni to więc, a po takim niespodziewanym zakończeniu widzimy w oknie błędów *`komunikat`*, jaki podaliśmy w dyrektywie `#error` (nie musi on koniecznie być ujęty w cudzysłowy, ale to dobry zwyczaj).

Dla ilustracji tego polecenia uzupełnimy piętrowy warunek `#if` z poprzedniego paragrafu:

```
#if PLATFORM == WINDOWS
   std::cout << "Kod kompilowany w systemie Windows";
#elif PLATFORM == LINUX
   std::cout << "Program budowany w systemie Linux";
// ...
```

```
#else
   #error "Nieznany system operacyjny, kompilacja przerwana!"
#endif
```

Jeżeli nie zdefiniujemy makra `PLATFORM` lub będzie miało inną wartość niż podane stałe `WINDOWS`, `LINUX`, itd., to preprocesor zareaguje odpowiednim błędem. W Visual C++ .NET wygląda on tak:

```
fatal error C1189: #error :  "Nieznany system operacyjny, kompilacja przerwana!"
```

Jak widać jest to „błąd fatalny", który zawsze powoduje przerwanie kompilacji programu.

***

W ten oto sposób zakończyliśmy omawianie dyrektyw preprocesora, służących kontroli procesu kompilacji programu. Obok makr jest to najważniejszy aspekt zastosowania mechanizmu wstępnego przetwarzania kodu.
Te dwa tematy nie są aczkolwiek pełnią możliwości preprocesora. Teraz poznamy jeszcze kilka dyrektyw ogólnego przeznaczenia - nie mniej ważnych niż te dotychczasowe.

Reszta dobroci

Pozostałe dyrektywy preprocesora są także bardzo istotne. Jedna z nich jest na tyle kluczowa, że widzisz ją w każdym programie napisanym w C++.

Dołączanie plików

Tą dyrektywą jest oczywiście `#include`. Już przynajmniej dwa razy przyglądaliśmy się jej bliżej, lecz teraz czas na wyjaśnienie wszystkiego.

Dwa warianty `#include`

Zaczniemy od przypomnienia składni tej dyrektywy. Jak wiemy, istnieją jej dwa warianty:

```
#include <nazwa_pliku>
#include "nazwa_pliku"
```

Oba powodują dołączenie pliku o wskazanej nazwie. Podczas przetwarzania kodu preprocesor usuwa po prostu wszystkie dyrektywy `#include`, wstawiając na ich miejsce zawartość wskazywanego przez nie plików.
Dzieki temu, że robi to preprocesor, a nie my, zyskujemy na kilku sprawach:

- nasze pliki kodu nie są (zbyt) duże, bo zawartość dołączanych plików (nagłówkowych) nie jest w nich wstawiona na stałe, a jedynie dołączana na czas kompilacji
- chcąc zmienić zawartość współdzielonych plików, nie musimy modyfikować ich kopii we wszystkich modułach, które zeń korzystają
- mamy więcej czasu, a przecież czas to pieniądz ;D

Skoro zaś `#include` oddaje nam tak cenne usługi, pomówmy o jej dwóch wariantach i różnicach między nimi.

Z nawiasami ostrymi

Model z nawiasami ostrymi (tworzonymi poprzez znak mniejszości i większości):

```
#include <nazwa_pliku>
```

stosowaliśmy od samego początku nauki C++. Nieprzypadkowo: pliki, jakie dołączamy w ten sposób, są po prostu niezbędne do wykorzystania niektórych elementów języka, Biblioteki Standardowej oraz innych bibliotek (Windows API, DirectX, itd.).

Gdy preprocesor widzi dyrektywę `#include` w powyższej postaci, to zaczyna szukać podanego pliku w jednym z wewnętrznych katalogów kompilatora, gdzie znajdują się pliki dołączane (ang. *include files*). Takich katalogów jest zwykle kilka, więc preprocesor przeszukuje ich listę; foldery te zawierają m.in. nagłówki Biblioteki Standardowej C++ (*string*, *vector*, *list*, *ctime*, *cmath*, …), starszej Biblioteki Standardowej C (*time.h*, *math.h*, …[103]), a często także nagłówki innych zainstalowanych bibliotek.

Chcąc przejrzeć lub zmodyfikować listę katalogów z plikami dołączanymi w Visual C++ .NET, musimy wybrać z menu *Tools* pozycję *Options*. Dalej przechodzimy do zakładki *Projects*|*VC++ Directories*, a na liście rozwijalnej *Show directories for:* wybieramy *Include files*.

Z cudzysłowami

Drugi typ instrukcji `#include` wygląda następująco:

```
#include "nazwa_pliku"
```

Z nimtakże zdążyliśmy się już spotkać - stosowaliśmy go do włączania własnych plików nagłówkowych do swoich modułów.

Ten wariant `#include` działa w sposób nieco bardziej kompleksowy niż poprzedni. Wpierw bowiem przeszukuje on bieżący katalog - tzn. ten katalog, w którym umieszczono plik zawierający dyrektywę `#include`. Jeśli tam nie znajdzie podanego pliku, wówczas zaczyna zachowywać się tak, jak `#include` z nawiasami ostrymi. Przegląda więc zawartość katalogów z listy folderów plików dołączanych.

Który wybrać?

Dwa rodzaje jednej dyrektywy to całkiem sporo. Którą wybrać w konkretnej sytuacji?…

Nasz czy biblioteczny

Decyzja jest jednak bardzo prosta:

- jeżeli dołączamy nasz własny plik nagłówkowy - taki, który znajduje się gdzieś blisko, na przykład w tym samym katatlogu - to powinniśmy skorzystać z dyrektywy `#include`, podając nazwę pliku **w cudzysłowach**
- jeśli natomiast wykorzystujemy nagłówek biblioteczny, pochodzący od kompilatora czy innych związanych z nim komponentów - stosujmy `#include` **z nawiasami ostrymi**

Teoretycznie można byc zawsze stosować wariant z cudzysłowami. To jednak obniżałoby czytelnośc kodu, gdyż nie można byłoby łatwo odróżnić, które dyrektywy dołączają nasze własne nagłówki, a które - nagłówki biblioteczne. Lepiej więc stosować rozróżnienie.

Nie pisałem tego na początku tej sekcji, ale chyba wiesz doskonale (bo mówiłem o tym wcześniej), że poprawne jest dołączanie **wyłącznie plików nagłówkowych**. Są to pliki zawierające deklaracje (prototypy) funkcji nie-inline, definicje funkcji *inline*, deklaracje

[103] Te nagłówki sa niezalecane, należy stosować ich odpowiedniki bez rozszerzenia .h i literką '*c*' na początku. Zamiast np. *math.h* używamy więc *cmath*.

zapowiadające zmiennych oraz definicje klas (a często także definicje szablonów, ale o tym później). Pliki te mają zwykle rozszerzenie *.h*, *.hh*, lub *.hpp*.

Ścieżki względne

W obu wersjach `#include` możemy wykorzystywać tzw. **ścieżki względne** (ang. *relative paths*), choć prawdziwie przydatne są one tylko w dyrektywie z cudzysłowami.

Ścieżki względne pozwalają dołączać pliki znajdujące się w innym katalogu niż bieżący[104]: w podkatalogach lub w nadkatalogu czy też w innych katalogach tego samego poziomu. Oto kilka przykładów:

```
#include "gui\buttons.h"              // 1
#include "..\base.h"                  // 2
#include "..\common\pointers.hpp"     // 3
```

Dyrektywa `1` powoduje dołączenie pliku *buttons.h* z podkatalogu *gui*. Kolejne użycie `#include` dołączy nam plik *base.h* z katalogu nadrzędnego względem obecnego. Z kolei ostatnia dyrektywa powoduje wpierw wyjście z aktualnego katalogu (`..`), następnie wejście do podkatalogu *common*, pobranie zeń zawartości pliku *pointers.hpp* i wstawienie w miejsce linijki `3`.

Jak widać, w `#include` można wykorzystać te same zasady tworzenia ścieżek względnych, jakie obowiązują w całym systemie operacyjnych[105].

Zabezpieczenie przed wielokrotnym dołączaniem

Dyrektywa `#include` jest głupia jak cały preprocesor. Ona tylko wstawia tekst podanego w pliku w miejsce swego wystąpienia. Nie dba przy tym, czy takie wstawienie spowoduje jakieś niepożądane efekty. A łatwo może przecież takie skutki wywołać…
Wyobraźmy sobie, że dołączamy plik A, który sam dołącza plik B i X. Niech plik B też dołącza plik X i już mamy problem: ewentualne definicje zawarte w X będą przez kompilator odczytane dwukrotnie. Zareaguje on wtedy błędem.

Trzeba więc podjąć ku temu jakiś środki zaradcze.

Tradycyjne rozwiązanie

Rozwiązanie problemu znanym jeszcze z C jest zastosowanie kompilacji warunkowej. Musimy po prostu objąć cały plik nagłówkowy (nazwijmy go *plik.h*) w dyrektywy `#ifndef`-`#endif`:

```
#ifndef _PLIK__H_
#define _PLIK__H_

// (cała treść pliku nagłówkowego)

#endif
```

Użyte tu makro (`_PLIK__H_`) powinno być najlepiej spreparowane w jakiś sposób z nazwy i rozszerzenia pliku - a jeśli trzeba, także i ze ścieżki do niego.

---

[104] Bieżący - to znaczy ten katalog, gdzie znajduje sie plik z dyrektywą `#include "..."`.
[105] Jako separatora możemy użyć slasha lub backslasha. Slash ma tę zaletę, że działa także w systemach unixowych - jeśli oczywiście dla kogoś jest to zaletą…

Jak to działa? Otóż dyrektywa `#ifndef` przepuści tylko jedno wstawienie treści pliku. Przy powtórnej próbie makro `_PLIK__H_` bedzie już zdefiniowane, więc cała zawartość pliku zostanie wyłączona z kompilacji.

Pomaga kompilator

Zaprezentowany wyżej sposób ma przynajmniej kilka wad:

- wymaga wymyślania nazwy dla makra kontrolnego, co przy dużych projektach, gdzie łatwo występują nagłówki o tych samych nazwach, staje się kłopotliwe. Sytuacja wygląda jeszcze gorzej w przypadku bibliotek pisanych przez nas: tam makra powinni mieć w nazwie także określenie biblioteki, aby nie prowokować potencjalnych konfliktów z innymi zasobami kodu
- umieszczona na końcu pliku dyrektywa `#endif` może być łatwo przeoczona i omyłkowo skasowana. Nietrudno też napisać jakiś kod poza klamrą `#ifndef`-`#else` - on nie będzie już objęty ochroną
- „sztuczka" wymaga aż trzech linii kodu, w tym jednej umieszczonej na samym końcu pliku

Mnie osobiście rozwiązanie to wydaje się po prostu nieeleganckie - zwłaszcza, że coraz więcej kompilatorów oferuje inny sposób. Jest nim umieszczenie gdziekolwiek w pliku dyrektywy:

```
#pragma once
```

Jest to wprawdzie polecenie zależne od kompilatora, ale obsługiwane przez wszystkie liczące się narzędzia (w tym także Visual C++ .NET oraz kompilator GCC z Dev-C++). Jest też całkiem prawdopodobne, że taka metoda rozwiązania problemu wielokrotnego dołączania znajdzie się w końcu w standardzie C++.

Polecenia zależne od kompilatora

Na koniec omówimy sobie takie polecenia, których wykonanie jest zależne od kompilatora, jakiego używamy.

Dyrektywa `#pragma`

Do wydawania tego typu poleceń służy dyrektywa `#pragma`:

```
#pragma polecenie
```

To, czy dane *`polecenie`* zostanie faktycznie wzięte pod uwage podczas kompilacji, zależy od posiadanego przez nas kompilatora. Preprocesor zachowuje się jednak bardzo porządnie: jeśli stwierdzi, że dana komenda jest nieznana kompilatorowi, wówczas cała dyrektywa zostanie po prostu zignorowana. Niektóre troskliwe kompilatory wyświetlają ostrzeżenie o tym fakcie.

Po opis poleceń, jakie są dostępne w dyrektywie `#pragma`, musisz udać się do dokumentacji swojego kompilatora.

Ważniejsze parametry `#pragma` w Visual C++ .NET

Używający innego kompilatora niż Visual C++ .NET mogą opuścić ten paragraf.

Ponieważ zakładam, że większość czytelników używa zalecanego na samym początku kursu kompilatora Visual C++ .NET, sądzę, że pożyteczne będzie przyjrzenie się kilku parametrom dyrektywy `#pragma`, jakie są tam dostępne.

Nie omówimy ich wszystkich, gdyż nie jest to podręcznik VC++, a poza tym wiele z nich dotyczy sprawa bardzo niskopoziomowych. Przypatrzymy się aczkolwiek tym, które mogą być przydatne przeciętnemu programiście.

Opisy wszystkich parametrów dyrektywy `#pragma` w Visual C++ .NET możesz rzecz jasna znaleźć w dokumentacji MSDN.

Wybrane parametry podzieliłem na kilka grup.

Komunikaty kompilacji

Pierwsza trójka parametrów `#pragma` pozwala na wyświetlanie pewnych informacji podczas procesu kompilacji programu. W przeciwieństwie do `#error`, polcenia nie powoduje jednak przerwania tego procesu, lecz tylko pełnią funkcję powiadamiającą np. o pewnych decyzjach podjętych w czasie kompilacji warunkowej.

Przyjrzyjmy się tym komendom.

message

Składnia polecenia message jest następująca:

```
#pragma message("komunikat")
```

Gdy preprocesor napotka powyższą linijkę kodu, to wyświetli w oknie komunikatów kompilatora (tam, gdzie zwykle pdoawane są błędy) wpisany tutaj *komunikat*. Jego wypisanie nie spowoduje jednak przerwania procesu kompilacji, co różni `#pragma message` od dyrektywy `#error`.

Przykładowym użyciem tego polecenie może być piętrowy `#if` podobny do tego z jakim mieliśmy do czynienia w poprzednim podrozdziale:

```
#define KEYBOARD     1
#define MOUSE        2
#define TRACKBALL    3
#define JOYSTICK     4

#define INPUT_DEVICE KEYBOARD

#if (INPUT_DEVICE == KEYBOARD)
   #pragma message("Wkompilowuje obsluge klawiatury")
#elif (INPUT_DEVICE == MOUSE)
   #pragma message("Domylsne urzadzenie: mysz")
#elif (INPUT_DEVICE == TRACKBALL)
   #pragma message("Sterowanie trackballem")
#elif (INPUTDEVICE == JOYSTICK)
   #pragma message("Obsluga joysticka")
#else
   #error "Nierozpoznane urzadzenie wejsciowe!"
#endif
```

Teraz, w zależnie od wartości makra `INPUT_DEVICE` w polu komunikatów kompilatora zobaczymy na przykład:

```
Sterowanie trackballem
```

W parametrze `message` możemy też stosować makra, np.:

```
#pragma message("Kompiluje modul " __FILE__ ", ktory byl ostatnio " \
```

```
                "zmodyfikowany: " __TIMESTAMP__)
```

W ten sposób zobaczymy oprócz nazwy kompilowanego pliku także datę i czas jego ostatniej modyfikacji.

deprecated

Nieco inne zastosowanie ma parametr deprecated, lecz także służy do pokazywania komunikatów dla programisty podczas kompilacji. Oto jego składnia:

```
#pragma deprecated(nazwa_1 [, nazwa_2, ...])
```

`deprecated` znaczy dosłownie 'potępiony' i jest trochę zbyt teatralna, ale adekwatna nazwa dla tego parametru dyrektywy `#pragma`. `deprecated` pozwala na wskazanie, które nazwy w programie (funkcji, zmiennych, klas, itp.) są przestarzałe i nie powinny być używane. Jeżeli zostaną one wykorzystane w kodzie, wówczas kompilator wygeneruje ostrzeżenie.

Spójrzmy na ten przykład:

```
// ta funkcja jest przestarzała
void Funkcja()
{
   std::cout << "Mam juz dluga, biala brode...";
}
#pragma deprecated(Funkcja)

int main()
{
   Funkcja();        // spowoduje ostrzeżenie
}
```

W powyższym przypadku zobaczymy ostrzeżenie w rodzaju:

```
warning C4995: 'Funkcja': name was marked as #pragma deprecated
```

Zauważmy, że dyrektywę `#pragma deprecated` umieszczamy po definicji przestarzałego symbolu. W przeciwnym razie sama definicja spowodowałaby wygenerowanie ostrzeżenia.
Innym sposobem oznaczenia symbolu jako przestarzały jest poprzedzenie jego deklaracji frazą `__declspec(deprecated)`.

Możemy też oznaczać makra jako przestarzałe, lecz aby uniknąć ich rozwinięcia w dyrektywie `#pragma`, należy ujmować ich nazwy w cudzysłowy.

warning

Ten parametr nie generuje wprawdzie żadnych komunikatów, ale pozwala na sprawowanie kontroli nad tym, jakie ostrzeżenia są generowae przez kompilator.
Oto składnia dyrektywy `#pragma warning`:

```
#pragma warning(specyfikator_1: numer_1_1 [numer_1_2 ...]      \
                [; specyfikator_2: numer_2_1 [numer_2_2 ...]])
```

Wygląda ona dość skomplikowanie, ale w praktyce stosuje się tylko jeden specyfikator na każde użycie dyrektywy, więc właściwa postać staje się prostsza.

Co dokładnie robi `#pragma warning`? Otóż pozwala ona zmienić sposób traktowania przez kompilator ostrzeżeń o podanych *numerach*. Podejmowane działania określa dokładnie *specyfikator*:

specyfikator	***znaczenie***
`disable`	Powoduje wyłączenie raportowania podanych numerów ostrzeżeń. Sytuacje, w których powinny wystąpić, zostaną po prostu zignorowane, a programista nie będzie o nich powiadamiany.
`once`	Sprawia, że podane ostrzeżenia będą wyświetlane tylko raz, przy pierwszym wystąpieniu powodujących je sytuacji.
`default`	Przywraca sposób obsługi ostrzeżeń do trybu domyślnego.
`error`	Sprawia, że podane ostrzeżenia będą traktowane jako błędy. Ich wystąpienie spowoduje więc przerwanie kompilacji.

`1`
`2`
`3`
`4`
	Zmienia tzw. poziom ostrzeżenia (ang. *warning level*). Generalnie wyższy poziom oznacza mniejszą dolegliwość i niebezpieczeństwo ostrzeżenia. Przesunięcie danego ostrzeżenia do określonego poziomu powoduje, że jego interpretacja (wyświetlanie, przerwanie kompilacji, itd.) zależeć będzie od ustawień kompilatora dla danego poziomu ostrzeżeń. Za ustawienia te nie odpowiada jednak `#pragma warning`.

Tabela 15. Specyfikatory kontroli ostrzeżeń dyrektywy `#pragma warning` w Visual C++ .NET

Skąd natomiast wziąć numer ostrzeżenia?… Jest on podawany w komunikacie kompilatora - jest to liczba poprzedzona literą 'C', np.:

```
warning C4101: 'nZmienna' : unreferenced local variable
```

Do `#pragma warning` podajemy numer już bez tej litery. Chcąc więc wyłączyć powyższe ostrzeżenie, stosujemy dyrektywę:

```
#pragma warning(disable: 4101)
```

Pamiętajmy, że stosuje się on do wszystkich instrukcji po swoim wystąpieniu - podobnie jak wszystkie inne dyrektywy preprocesora.

Uwaga: jakkolwiek wyłączanie ostrzeżeń jest czasem konieczne, nie należy z tym przesadzać. Przede wszystkim nie wyłączajmy wszystkich pojawiających się ostrzeżeń „jak leci", lecz wpierw przyjrzyjmy się, jakie kod je powoduje. Każde użycie `#pragma warning(disable: numer)` powinno być bowiem dokładnie przemyślane.

Funkcje inline

Z poznanymi w tym rozdziale funkcjami *inline* jest związanych kilka parametrów dyrektywy `#pragma`. Zobaczmy je.

auto_inline

`#pragma auto_inline` ma bardzo prostą postać:

```
#pragma auto_inline([on/off])
```

Parametr ten kontroluje automatyczne rozwijanie krótkich funkcji przez kompilator. Ze względów optymalizacyjnych niektóre funkcje mogą być bowiem traktowane jako inline nawet wtedy,gdy nie są zadeklarowane z przydomkiem inline.
Jeśli z jakichś powodów nie chcemy aby tak było, możemy to wyłączyć:

```
#pragma auto_inline(off)
```

Wszystkie następujące dalej funkcje na pewno nie będą rozwijane w miejscu wywołania - chyba że sami tego sobie życzymy, deklarując je jako inline.

Typowo `#pragma auto_inline` stosujemy dla pojedynczej funkcji w ten sposób:

```
#pragma auto_inline(off)
void Funkcja(/* ... */)
{
   // ...
}
#pragma auto_inline(on)
```

Jeżeli nie podamy w dyrektywie ani `on`, ani `off`, to stan `auto_inline` zostanie zamieniony na przeciwny (z `on` na `off` lub odwrotnie).

inline_recursion

Ta komenda jest także przełącznikiem:

```
#pragma inline_recursion([on/off])
```

Kontroluje ona rozwijanie wywołań rekurencyjnych (ang. *recursive calls*) w funkcjach typu *inline*. **Rekurencją** (ang. *recurrency*) nazywamy zjawisko, kiedy jakaś funkcja wywołuje samą siebie - oczywiście nie zawsze, lecz w zależności od spełnienia jakichś warunków. Wywołania rekurencyjne są prostym sposobem na tworzenie pewnych algorytmów - szczególnie takich, które operują na rekurencyjnych strukturach danych, jak drzewa. Rekurencja może być bezpośrednia - gdy funkcja sama wywołuje siebie - lub pośrednia - jeśli robi to inna funkcja, wywołana wcześniej przez tą naszą.

Rekurencyjne mogą być także funkcje *inline*. W takim wypadku kompilator domyślnie rozwija tylko ich pierwsze wywołanie; dalsze wywołania rekurencyjne są już dokonywane w sposób właściwy dla normalnych funkcji.
Można to zmienić, powodując rozwijanie także dalszych przywołań rekurencyjnych (w ograniczonym zakresie oczywiście) - należy wprowadzić do kodu dyrektywę:

```
#pragma inline_recursion(on)
```

Łatwo się domyslić, że `inline_recursion` jest domyślnie ustawiona na `off`.

inline_depth

Z poprzednim poleceniem związane jest także to - dyrektywa `#pragma inline_depth`:

```
#pragma inline_depth(głębokość)
```

głębokość może tu być stałą całkowitą z zakresu od zera do `255`. Liczba ta precyzuje, jak głęboko kompilator ma rozwijać rekurencyjne wywołania funkcji inline. Naturalnie, wartośc ta ma jakiekolwiek znaczenie tylko wtedy, gdy ustawimy `inline_recursion` na `on`. Ponadto wartość `255` oznacza rozwijanie rekurencji bez ograniczeń (z wyjątkiem rzecz jasna zasobów dostępnych dla kompilatora).

Domyślnie rozwijanych jest osiem rekurencyjnych wywołań *inline*. Pamiętajmy, że przesada z tą wartością może dość łatwo doprowadzaić do rozrostu kody wynikowego - zwłaszcza, jeśli przesadzamy też z obdzielaniem funkcji modyfikatorami `inline` (a szczególnie `__forceinline`).

Inne

Oto dwie ostatnie komendy `#pragma` w Visual C++, jednak wcale nie są one najmniej ważne. Jakby to powiedzieli Anglicy, one są '*last but not least*' :) Przyjrzymy się im.

comment

To polecenia umożliwa zapisanie pewnych informacji w wynikowym pliku EXE:

```
#pragma comment(typ_komentarza [, "komentarz"])
```

Umieszczone tak komentarze nie służą naturalnie tylko do dekoracji (choć niektóre do tego też :D), lecz moga nieść także dane ważne dla kompilatora czy linkera. Wszystko zależy od frazy typ_komentarza. Oto dopuszczalne możliwości:

typ komentarza	***znaczenie***
`exestr`	Umieszcza w skompilowanym pliku tekstowy *komentarz*, który linker w niezmienionej postaci przenosi do konsolidowanego pliku EXE. Napis ten nie jest ładowany do pamięci podczas uruchamiania programu, niemniej istnieje w pliku wykonywalnym i można go odczytać specjalnymi aplikacjami.
`user`	Wstawia do skompilowanego pliku podany *komentarz*, jednak linker ignoruje go i nie pojawia się on w wynikowym EXEku. Istnieje natomiast w skompilowanym pliku *.obj*.
`compiler`	Dodaje do skompilowanego modulu informację o wersji kompilatora. Nie pojawia się ona wynikowym pliku wykonywalnym z programem. Przy stosowaniu tego typu, nie należy podawać żadnego *komentarza*, bo w przeciwnym razie kompilator uraczy nas ostrzeżeniem.
`lib`	Ten typ pozwala na podanie nazwy pliku **statycznej biblioteki** (ang. *static library*), która będzie linkowana razem ze skompilowanymi modułami naszej aplikacji. Linkowanie dodatkowych bibliotek jest często potrzebne, aby skorzystać z niestandardowego kodu, np. Windows API, DirectX i innych.
`linker`	Tak możemy podać dodatkowe opcje dla linkera, niezależnie od tych podanych w ustawieniach projektu.

Tabela 16. Typy komentarzy w dyrektywie `#pragma comment` w Visual C++ .NET

Spośród tych możliwości najczęściej stosowane są `lib` i `linker`, ponieważ pozwalają zarządzać procesem linkowania. Oprócz tego `exestr` umożliwia zostawienie w pliku EXE dodatkowego tekstu informacyjnego, np.:

```
#pragma comment(exestr, "Skompilowano: " __DATE__ __TIME__)
```

Jak widać na załączonym obrazku, w takim tekście można stosować też makra.

once

Na ostatku przypomnimy sobie pierwsze poznane polecenie `#pragma - once`:

```
#pragma once
```

Wiemy już doskonale, jakie jest działanie dyrektywy `#pragma once`. Otóż powoduje ona, że zawierający ją plik bedzie włączany tylko raz podczas przeglądania kodu przez preprocesor. Każde sukcesywne wystąpienie dyrektywy `#include` z tymże plikiem zostanie zignorowane.

Dyrektywa `#pragma once` jest obecnie obsługiwana przez bardzo wiele kompilatorów - nie tylko przez Visual C++. Istnieje więc niemała szansa, że niedługo podobna dyrektywa stanie się częścią standardu C++. Na pewno jednak nie będzie to `#pragma once`, gdyż wszystkie szczegóły dyrektyw `#pragma` są z założenia przynależne konkretnemu kompilatorowi, a nie językowi C++ w ogóle.

Jeśli sam miałbym optować za jakąś konkretną, ustandaryzowaną propozycją składniową dla tego rozwiązania, to chyba najlepsze byłoby po prostu `#once`.

***

I tą sugestią dla Komitetu Standaryzacyjnego C++ zakończyliśmy omawianie preprocesora i jego dyrektyw :)

Podsumowanie

Ten rozdział był poświęcony rzadko spotykanej w językach programowania właściwości C++, jaką jest preprocesor. Mogłeś z niego dowiedzieć się wszystkiego na temat roli tego ważnego mechanizmu w procesie budowania programu oraz poznać jego dyrektywy. Pozwoli ci to na sterowanie procesem kompilacji własnego kodu.

W tym rozdziale starałem się też w jak najbardziej obiektywny sposób przedstawić makra i makrodefinicje, gdyż na ich temat wygłasza się często wiele błędnych opinii. Chciałem więc uświadomić ci, że chociaż większość dawnych zastosowań makr została już wyparta przez inne konstrukcje języka, to makra są nadal przydatne w skracaniu zapisu czesto występujących fragmentów kodu oraz przede wszystkim - w kompilacji warunkowej. Istnieje też wiele sposobów na wykorzystanie makr, które noszą znamiona „trików” - być może natrafisz na takowe podczas lektury innych kursów, książek i dokumentacji. Warto byś wtedy pamiętał, że w stosowaniu makr, jak i we wszystkim w programowaniu, należy zawsze umieć znaleźć równowagę między efektownością a efektywnością kodowania.

Preprocesor oraz omówione wcześniej wskaźniki były naszym ostatnim spotkaniem z krainą starego C w obrębie królestwa C++. Kolejne trzy rozdziały skupiają się na zaawansowanych cechach tego ostatniego: programowaniu obiektowym (ze szczególnym uwzględnieniem przeciążania operatorów), wyjątkach oraz szablonach. Wpierw zobaczymy usprawnienia OOPu, jakie oferuje nam język C++.

Pytania i zadania

Możesz uważać, że preprocesor jest reliktem przeszłości, ale nie uchroni cię to od wykonania obowiązkowej pracy domowej! ;))

Pytania

1. Czym jest preprocesor? Kiedy wkracza do akcji i jak działa?
2. Na czym polega mechanizm rozwijania i zastępowania makr?
3. Jakie dwa rodzaje makr można wyróżnić?
4. Na jakie problemy można natrafić, jeżeli spróbuje się zastosować makra zamiast bardziej odpowiednich, innych konstrukcji języka C++?
5. Jakie dwa zastosowania makr pozostają nadal aktualne?
6. Jakie wyrażenia może zawierać warunek kompilacji dyrektyw `#if` i `#elif`?
7. Czym różnią się dwa warianty dyrektywy `#include`?
8. Jaką rolę pełni dyrektywa `#pragma`?

Ćwiczenia

1. Opracuj (klasyczne już) makro wyznaczające większą z dwóch podanych wartości.
2. (**Trudniejsze**) Odszukaj definicję klasy `CIntArray` z rozdziału o wskaźnikach i przy pomocy preprocesora przerób ją tak, aby można by z niej korzystać dla dowolnego typu danych.
3. Otwórz kod aplikacji rozwiązującej równania kwadratowe, którą (mam nadzieję) napisałeś w rozdziale 1.4. Dodaj do niej kod pomocniczy, wyświetlający wartość delta dla podanego równania; niech kompiluje się on tylko wtedy, gdy zdefiniowana zostanie nazwa `DEBUG`.
4. (**Trudne**) Skonstruuj warunek kontrolowanej kompilacji, który pozwoli na wykrycie platform 16-, 32- i 64-bitowych.
   Wskazówka: wykorzystaj charakterystykę typu `int`…