Exemplo de Tipos de Dados Criados pelo Programador com Compilação Separada

Neste diretório há dois exemplos de tipos de dados criados pelo usuário (programador). O primeiro tipo é o TALUNO. A definição desse tipo e das operações que são possíveis com ele estão no arquivo.h. A implementação das funções declaradas em aluno.h são feitas no arquivo aluno.c
O outro tipo é o tipo TURMA para o qual também há dois pares de arquivos: turma.h e turma.c.
Esse exemplo de programa modular mostra uma limitação da linguagem C em esconder a implementação do usuário (outros programadores que venham a usar os tipos). O problema é que o tipo TURMA, usa o tipo TALUNO. Em termos de representação de dados, um variável do tipo TURMA é um array do tipo TALUNO. Com isso não podemos fazer como nos exemplos nos slides, isto é, declarar a struct aluno em aluno.c e apenas deixar em aluno.h o typedef struct aluno TALUNO, porque se fizermos isso o compilador ao tentar compilar turma.c não encontra a definição da struct aluno e gera erro sempre que as funções implementadas em turma.c acessarem um elemento do vetor de TALUNO. Repare que não há como esconder dos usuários dos dois tipos a definição da struct aluno. Por isso ela teve que ser declarada em aluno.h, ficando os seus campos constituintes expostos, o que não é desejável para a implementação do tipo abstrato TALUNO. Mas isso é uma limitação da linguagem. Na linguagem C++ isso é contornado definindo-se TALUNO como uma *classe* (*class* no C++). Ou seja, no C++
é sempre possível esconder não apenas a implementação das operações, mas também, como os dados (structs, vetores, etc) formam o tipo. Na linguagem C, sempre é possível esconder a implementação das operações (funções) associadas ao tipo. Isso é feito por meio de compilação separada. O usuário do tipo tem acesso ao arquivo .h e o arquivo objeto com a implementação das operações, resultante da compilação do arquivo .c correspondente. Entretanto ele não tem acesso ao arquivo .c, logo não sabe como as operações são implementadas.

Compilação separada

A compilação separada pode ser feita de três formas distintas:
1) Compilação manual de cada módulo.
2) Compilação através do uso de um arquivo Makefile e do “make” (um comando da shell do sistema operacional.
3) Criando-se um projeto em uma IDE - *Integrated Development Environment* (como Geany, DropBox, etc).

Compilação manual

A compilação manual de cada módulo pode ser feita com o gcc do seguinte modo:

gcc -o tipo.o -Wall -c tipo.c

Exemplo: nesse diretório temos os tipos TALUNO e TURMA, cujas operações estão implementadas, respectivamente em aluno.c e turma.c. Podemos gerar os programas objetos para esses arquivos com os comandos:

gcc -o aluno.o -Wall -c aluno.c

e

gcc -o turma.o -Wall -c turma.c

Um programador usuário desses tipos precisa ter apenas os arquivos .h correspondentes (**aluno.h** e **turma.h**) e também os arquivos .o (**aluno.o** e **turma.o**). O programador cria seu próprio arquivo com a função main() que faz uso desses tipos, incluindo no seu arquivo os dois arquivos headers (**aluno.h** e **turma.h).** Em seguida, o usuário pode gerar o arquivo objeto do seu programa, em linguagem de máquina. Vamos supor que o arquivo criado pelo usuário tem nome **main.c.** Ele pode compilar esse arquivo gerando o módulo objeto, com o comando:

gcc -o main.o -Wall -c mai.c

Esse arquivo (**main.o)** ainda não é executável. Ele tem apenas a tradução para linguagem de máquina do arquivo que contém a função principal *main()* e, possivelmente, outras funções criadas pelo usuário dos tipos. Para se ter um módulo executável, é necessário "ligar" todos os arquivos objetos. Isso é feito através de um *linkeditor.* O linkeditor do gcc pode ser ativado através do seguinte comando:

gcc modulo1.o modulo2.o .... modulomain.o -o executavel

Repare que um e apenas um dos módulos em linguagem de máquina (arquivos. o) devem conter a função main. Por exemplo, para gerar manualmente o programa executável para os módulos desse diretório, basta executar em uma shell o comando:

gcc aluno.o turma.o main.o -o main

Esse comando gera o arquivo executável **main**. Para testar esse programa, você pode passar para ele o arquivo de entrada **entrada.txt** através do seguinte comando:

./main < entrada.txt

Compilando e ligando com o comando make

o comando make utiliza um arquivo de nome **Makefile** ou **makefile** que lhe dá instruções de como e quando compilar e ligar módulos. A especificação de como construir um Makefile é complexa. Caso você se queira entender mais como fazer seu próprio Makefile, existem vários videos e tutoriais na internet. Nesse texto será explicado como o arquivo Makefile para o exemplo em questão funciona. Você vê dois tipos de linha no arquivo Makefile: linha de dependência e linha de comando. O tipo linha dependência tem o formato: *arquivo_alvo: arq_depen1 arq_depen2 ... arq_depenn.* A primeira linha do Makefile é um exemplo desse tipo de linha. Ela indica que o processamento para gerar arquivo_alvo (que será indicado em uma linha de comando) depende dos arquivos *arq_depen1 ... arq_depenn.* Se nenhum desses arquivos for modificado, não é necessário executar os comandos para gerar o *arq_alvo,* mas basta que um dos arquivos *arq_depen* seja modificado para que os comandos sejam executados para gerar o novo *arq_alvo.* Por exemplo, a primeira linha do arquivo Makefile contém

main: main.o aluno.o turma.o

ela informa que a geração do arquivo-alvo main depende da atualização de pelo menos um dos arquivos main.o, turma.o e turma.o

O segundo tipo de linha é uma linha de comando. Para uma mesma linha de dependência pode haver uma ou mais linhas de comando para se gerar o aquivo alvo. Cada linha de comando inicia com um caractere de tabulação e pode conter qualquer comando executável do sistema operacional, principalmente uma chamada ao compilador ou linkeditor para gerar o arquivo alvo. A segunda linha do arquivo Makefile desse diretório é um exemplo de linha de comando:

gcc main.o aluno.o turma.o -o main

Ela chama o linkeditor do gcc para ligar todos os arquivos objetos e gerar o executável **main** que é o arquivo alvo da primeira linha. Repare que no restante do arquivo Makefile, sempre há um par de linhas: linha de dependência, linha de comando. A linha de dependência pode ser utilizada para marcar um ponto de entrada no Makefile. Nesse caso o nome antes do sinal de ":" não é um arquivo

alvo, mas sim uma marca para se indicar o que se quer executar no Makefile. Um exemplo dessa situação é o último par de linhas do arquivo. A penúltima linha contém:

clean:

Essa linha tem a palavra *clean* e não tem arquivos que geram dependência (arq_depen). Essa linha não é executada em uma chamada normal ao comando make. Ela só é executada se o comando make vier seguido da palavra clean:

make clean

neste caso, o comando *rm *.o main* é executado, apagando do diretório todos os arquivos objetos e o executável main. Ele deve ser chamado para forçar que uma próxima chamada ao comando make reconstrua todos os arquivos-alvos.

Criando um projeto através de um ambiente de desenvolvimento (IDE)

Com o uso de uma IDE, fica bem mais fácil gerenciar a compilação de módulos e a criação do executável. Geralmente as IDES denominam de *projeto (project)* o conjunto de módulos de um sistema e possuem um módulo para criar e gerenciar os projetos. Internamente, as IDEs utilizam um Makefile com o make ou outra ferramenta semelhante para gerenciar a compilação e ligação dos módulos, mas escondem a complexidade desse trabalho do usuário. Cada IDE tem o seu próprio modo de oferecer o recurso de projeto, portanto é necessário que o programador usuário consulte a documentação da IDE para poder criar os seus projetos.