


Laboratório de Engenharia de Software

Aula 11

Implementação da programação modular 2

Arndt von Staa
LES/DI/PUC-Rio
Abril 2009

Especificação




Laboratório de Engenharia de Software

- Objetivo dessa aula
 - Mostrar como implementar em C os conceitos de interface única, preservando o controle de tipos dos elementos declarados nas interfaces entre módulos
 - Descrever o funcionamento da ferramenta **link**
- Referência básica:
 - Capítulo 6 do livro texto

Ago 2008Arndt von Staa © LES/DI/PUC-Rio2 / 34

Laboratório de Engenharia de Software

Sumário



- Declaração e definição
- Pilha de execução, registro de ativação
- Classes de memória
- Ligação
- Pré-processamento
- Padrão de programação C para a interface de módulos


Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

3 / 34

Laboratório de Engenharia de Software

O que ocorre ao declarar um elemento?



- Ao **declarar** um **nome** em uma **linguagem tipada** é estabelecido
 - que o nome existe
 - que corresponde a valores de um determinado tipo
 - isto permite **gerar código que interpreta corretamente** os valores acessados a partir deste nome

Ago 2008


Arndt von Staa © LES/DI/PUC-Rio

4 / 34

Laboratório de Engenharia de Software

O que ocorre ao definir um elemento?

- Ao **definir** um nome em uma linguagem tipada
 - é **alocado o espaço de dados** a ser ocupado pelo valor
 - a extensão do espaço **depende do tipo**
 - o nome é **amarrado** ao espaço
- Ao **somente declarar** um nome, **não é definido o espaço** de dados amarrado a esse nome
- Ao definir um nome pode-se **inicializar** o valor do espaço de dados, exemplos
 - `int x = 10 ;`
 - `char Mensagem[] = "texto da mensagem" ;`




Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
5 / 34

Laboratório de Engenharia de Software

Exemplos de declaração e definição

- De maneira geral, ao redigir o código de uma declaração ocorre tanto a declaração como a definição


```
int F( int x )
{
    int y = 10 ;
    ...
}
```
- **declara e define** **x** e **y**, sendo que **x** e **y** fazem parte do **registro de ativação** de **F** – (**automático**)
- gera o **código de inicialização** de **y** a ser executado imediatamente ao entrar na função **F**
- **declara, define e inicializa** **F**, sendo que o “valor” de **F** é o código e que faz parte do segmento executável (**estático**)



Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
6 / 34

Laboratório de Engenharia de Software

Pilha de execução

Topo da pilha de execução →

Y
X
ret
ant

Função F

Y
X
ret
ant

Função F

ret
ant

Função G

- Cada **registro de ativação C** contém:
 - *ant*: endereço do **topo da pilha** do registro de ativação anterior
 - *ret*: endereço de **retorno de execução** para após o local em que foi chamada
 - **espaços** para os **parâmetros e variáveis locais**
 - espaços para **temporários**
- Todos **endereço locais** são **deslocamentos relativos ao topo da pilha** de execução

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
7 / 34

Laboratório de Engenharia de Software

Exemplos de declaração e definição


- `int x ;` fora de uma função
 - **declara e define** **x** como **global externado** pelo módulo
- `static int Y ;` fora de uma função
 - **declara e define** **Y** como **global encapsulado** no módulo
- `extern int Z ;`
 - **somente declara** **Z** como **global externo** ao módulo
- `int G(float X) ;`
 - **somente declara** a função do tipo: `(float) : int`
- **Observação:** `static` para **variáveis locais** indica que
 - existe somente uma instância da variável, mesmo se existirem n chamadas recursivas na pilha de execução e
 - o valor é preservado de uma ativação para outra, mesmo se nenhum registro de ativação estiver na pilha de execução

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
8 / 34

4

Laboratório de Engenharia de Software

Restrições para definições




- Em C e C++
 - um **único módulo** do programa deve **declarar e definir** o nome compartilhado
 - **os demais** módulos devem somente **declarar**
- Isso vale para nomes aos quais se associam espaços
 - nomes de variáveis
 - nomes de funções
 - mas não para nomes de tipos
 - tipos são somente declarados
- Podem-se declarar nomes e não utilizá-los
- Quando se define um nome local e não o utiliza o compilador emite uma mensagem de advertência

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
9 / 34

Laboratório de Engenharia de Software


Classes de memória real



- **Executável**
 - é onde se armazena o **código a ser executado** e, possivelmente, algumas constantes numéricas
- **Estática encapsulada**
 - contém todos os espaços de dados **globais encapsulados** (declarados **static**), e todas as constantes literais definidas no interior de módulos
- **Estática visível**
 - contém todos os espaços de dados **globais externados**
- **Automática**
 - contém a **pilha de execução do programa**
- **Dinâmica**
 - contém espaços de dados alocados pelo usuário (**malloc**, **new**)

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
10 / 34

Outra classe de memória




Laboratório de Engenharia de Software

- **Persistente**
 - é onde se armazenam os dados que estarão disponíveis de uma **instância de execução** do programa para outra
 - arquivos contendo parâmetros de execução
 - tipicamente os arquivos **.dat** e **.ini**
 - bases de dados
 - segmentos de memória virtual

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
11 / 34

Ligação




Laboratório de Engenharia de Software

- A ligação **combina** $m \geq 1$ módulos objeto e módulos contidos em uma ou mais **bibliotecas**, produzindo o **programa carregável** (**.EXE**, **.COM**)
- No módulo objeto todos os **endereços gerados** são **deslocamentos** (*offsets*) relativos a zero dentro da respectiva **classe de memória**
- O **ligador** (*linker*) justapõe (concatena) os espaços de cada uma das classes de memória (**segmentos**: executável e estática) definidos nos módulos objeto, formando um único grande espaço para cada classe de memória
- O estes dois grandes **segmentos** constituem o **programa carregável**

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
12 / 34

Composição de um módulo objeto



Laboratório de Engenharia de Software

Dados estáticos

Código

Tab Reloc

Tab Simb

→ Declarados e definidos, relativos a 0


→ Relocável, relativo a 0

→ **Tabela de relocação**
- informa os locais contendo endereços a serem relocados

→ **Tabela de símbolos**
- referências a nomes externos declarados e não definidos (importados)
- referências a nomes externos declarados e definidos (externados pelo módulo)

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
13 / 34

Símbolos definidos no módulo objeto



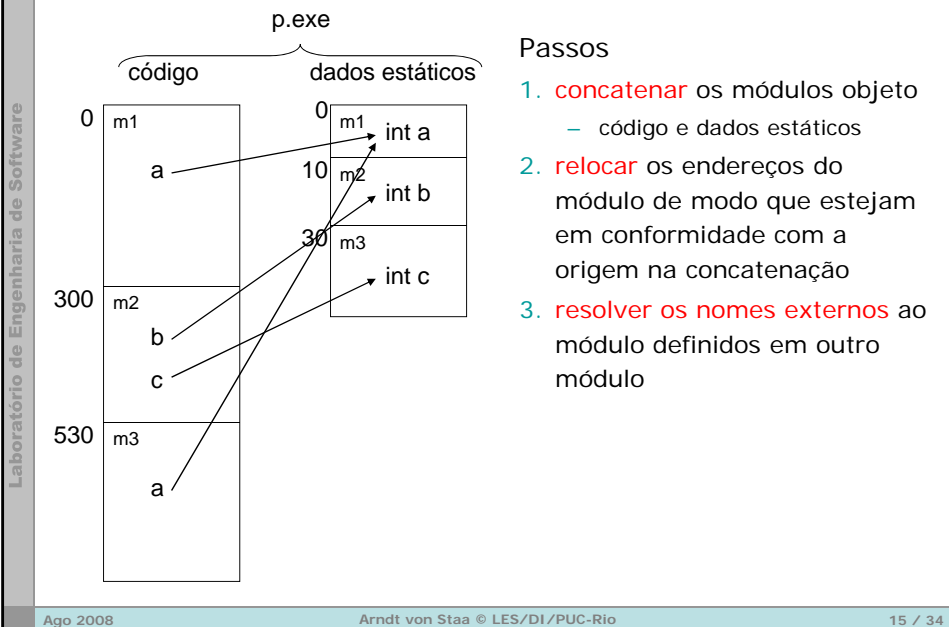
Laboratório de Engenharia de Software

- Cada módulo objeto contém uma **tabela de símbolos** agregando os nomes globais externos, particionada em
 - símbolos somente declarados
 - símbolos declarados e definidos
- Os símbolos somente declarados definem uma **lista de todos os locais** no código (ou nos dados) em que o símbolo é referenciado
- Um nome externo somente declarado em um determinado módulo necessariamente deverá estar declarado e definido em exatamente um outro módulo do programa sendo composto
 - i.e. cada nome deve ser **definido em um único módulo**

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
14 / 34

7

Composição de um executável




Relocação



- Laboratório de Engenharia de Software
- O ligador **ajusta os endereços** dos elementos contidos em cada segmento de modo que passem a ser deslocamentos relativos à **origem** dos correspondentes **segmentos** do programa
 - A relocação ocorre com relação aos segmentos
 - código
 - estático local e externo
 - A **tabela de relocação** contida no módulo objeto informa os pontos no código e nos dados globais do módulo que deverão ser ajustados
 - no módulo objeto os deslocamentos são relativos a zero
 - para relocar basta somar a origem do segmento do módulo definida no segmento composto às referências internas ao módulo registradas na tabela de relocação
- Ago 2008 Arndt von Staa © LES/DI/PUC-Rio 16 / 34

Laboratório de Engenharia de Software

Resolução de nomes externos, sem bibliotecas




- O ligador cria uma **tabela de símbolos** que conterá os **nomes externos**. Cada símbolo informa
 - o endereço no segmento composto
 - a lista dos locais que referenciam o símbolo ainda não definidos
- Ao encontrar um nome externo
 - adiciona-o à tabela caso ainda não figure lá
 - **se for** um nome externo **declarado e definido**
 - se a tabela de símbolos do ligador já define o nome, é emitido um **erro de duplicação de definição**
 - caso contrário, percorre a lista dos locais que referenciam o símbolo e atribui o endereço definido
 - **se for** um nome externo **somente declarado**
 - se a tabela de símbolos do ligador já define o nome, atribui esta definição aos locais no módulo que referenciam este símbolo
 - caso contrário, o ligador acrescenta a lista do módulo à lista do ligador
- Ao terminar o processamento
 - para cada símbolo não definido contido na tabela do ligador, é emitido um **erro de símbolo não definido**

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
17 / 34


Laboratório de Engenharia de Software

Resolução de nomes externos, com bibliotecas



- Uma **biblioteca estática (.lib)** é formado por
 - uma lista de módulos
 - uma tabela de símbolos contendo os símbolos externados pelos módulos e a referência para o código do respectivo módulo na lista de módulos
- Após compor todos os módulos objeto, para cada símbolo ainda não definido
 - o ligador procura este símbolo, segundo a ordem de fornecimento das bibliotecas
 - caso seja encontrado, o módulo correspondente é extraído da biblioteca e acrescentado ao programa sendo montado
 - para isso segue o procedimento anterior
 - caso não seja encontrado, é emitido um **erro de símbolo não definido**
 - repete até todos os símbolos terem sido processados


Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
18 / 34

Ligação dinâmica


Laboratório de Engenharia de Software

- **Bibliotecas dinâmicas** (.dll) são carregadas à medida que forem acessadas durante o processamento
 - ao encontrar um símbolo externo ainda não resolvido
 - utiliza a .dll, se já carregada, ou então carrega ela
 - substitui a referência ao símbolo para a referência à função
 - cada biblioteca é compartilhada por todos os programas que a usem
 - cada programa estabelece espaços próprios para os dados
- Vantagens
 - uma biblioteca dinâmica é carregada uma única vez considerando todos os programas em execução simultânea
 - pode-se trocar uma biblioteca sem precisar recompilar ou religar todo o programa

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
19 / 34

Ligação dinâmica


Laboratório de Engenharia de Software

- Problemas
 - precisa-se projetar com muito cuidado as bibliotecas dinâmicas, visando explicitamente a possibilidade do seu reuso em diversos programas
 - as bibliotecas são conhecidas pelo nome, portanto pode ocorrer colisão de nomes
 - bibliotecas diferentes com o mesmo nome
 - é necessário assegurar que a versão correta da biblioteca seja utilizada com cada um dos programas
 - todos os programas utilizam a mesma versão da biblioteca, a menos que se possa armazenar as bibliotecas em locais distintos
 - todos evoluem à medida que as bibliotecas forem evoluindo

Ago 2008
Arndt von Staa © LES/DI/PUC-Rio
20 / 34

Carga de um programa



- Para poderem ser executados programas **precisam estar em memória real**
 - **fragmentos** de um programa executável podem estar em qualquer um dos segmentos: executável, pilha (automático), estático, e dinâmico. A **origem** estará no segmento executável.
- Ao ativar um programa é ativado o **carregador** que recebe como parâmetro o **nome do arquivo** contendo o programa a ser carregado
- O carregador
 - determina **onde serão colocados** os segmentos executável e estático e copia os segmentos do arquivo para a memória
 - efetua as necessárias **relocações** de modo a ajustar os endereços contidos nesses segmentos
 - **ativa** o programa chamando a função **main()** (ou **winmain()**)

Ago 2008

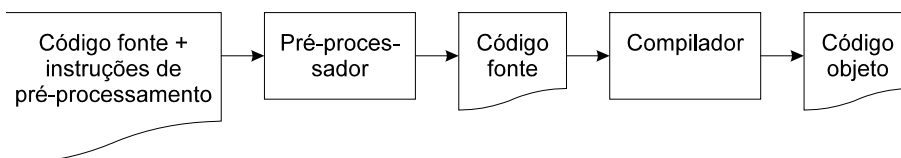
Arndt von Staa © LES/DI/PUC-Rio

21 / 34

Pré-processamento



- Um **pré-processador**
 - é um **processador de linguagem**
 - recebe um arquivo contendo texto **fonte** e **diretivas de pré-processamento**
 - produz um outro arquivo de texto fonte na mesma linguagem



Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

22 / 34

Pré-processamento em C / C++



- **#include <nome-arquivo>**
 - procura o arquivo no domínio do compilador e o inclui
- **#include "nome-arquivo"**
 - procura o arquivo no domínio do usuário e o inclui
- **#define NomeElem ValorElem**
 - define o nome **NomeElem**
 - provoca a substituição de todas as ocorrências de **NomeElem** no código por **ValorElem**
 - **ValorElem** pode ser o string nulo (vazio)
- **#undef NomeElem**
 - retira o nome **NomeElem** da tabela do pré-processador
- **#pragma**
 - permite informar coisas específicas ao compilador

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

23 / 34

Pré-processamento em C / C++



```
#if Expressão
    TextoTrue – é compilado se expressão != 0
#else
    TextoFalse – é compilado se expressão == 0
#endif

#ifdef Nome      ou ao contrário  #ifndef Nome
    TextoTrue – é compilado se Nome estiver definido
#else
    TextoFalse – é compilado se Nome não estiver definido
#endif
```

- Operador de pré-processamento: **defined(nome)**
- Outros: **#nome** converte nome para string
- **##nome** copia nome (usualmente um parâmetro de macro)

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

24 / 34

Padrão de programação C



- Ao desenvolver programas em C ou C++ siga o recomendado no apêndice 1 *Padrão de Composição de Módulos C e C++*.
- Todos os módulos que podem ser incluídos devem conter um controle de compilação única
 - módulo de definição
 - tabelas de definição
 - tabelas de dados

```
#if !defined( Nome-arquivo_MOD )
#define      Nome-arquivo_MOD
/* Comentário cabeçalho do arquivo */
    Corpo do arquivo
#endif
/* Comentário fim de arquivo*/
```

Consistência das interfaces, relembração



- Para **garantir a consistência** entre módulos cliente e módulos servidores, é fundamental que se utilize **exatamente a mesma definição de interface** tanto ao compilar o módulo servidor, como ao compilar os diversos módulos cliente deste módulo servidor
- O código da interface é redigido no módulo de definição (**.h**)
 - será incorporado através de **#include**
 - ao compilar o correspondente módulo de implementação
 - ao compilar módulos cliente deste módulo.

Consistência das interfaces C/C++



- Problema devido a propriedades sintáticas das linguagens C/C++
 - variáveis globais externas devem aparecer exatamente uma vez sem o declarador **extern**
 - todas as outras vezes devem vir precedidas deste declarador
- Pode-se conseguir isso com código de pré-processamento que assegure
 - sempre que um cliente compilar o módulo de definição, as declarações de variáveis globais externas estejam precedidas de **extern**
 - sempre que o próprio módulo compilar o módulo de definição, as variáveis globais externas não estejam precedidas de **extern**

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

27 / 34

Padrão para o módulo de definição



- Inicie o código do corpo do módulo de definição com o seguinte esquema de código:

```
/* Controle de escopo do arquivo de definição */
#ifdef Nome-arquivo-modulo_OWN
    #define Nome-arquivo-modulo_EXT
#else
    #define Nome-arquivo-modulo_EXT extern
#endif
```
- Ao final do código do módulo de definição coloque o código:

```
#undef Nome-arquivo-modulo_EXT
```

Ago 2008

Arndt von Staa © LES/DI/PUC-Rio

28 / 34

Padrão para o módulo de definição



- Declare cada **variável global externa** não inicializada da seguinte forma:

```
Nome-arquivo-modulo_EXT declaração-de-variável ;
```

- Declare cada variável global externa inicializada da seguinte forma

```
Nome-arquivo-modulo_EXT declaração-variavel
#ifdef Nome-arquivo-modulo_OWN
    = Inicialização ;
#else
    ;
#endif
```

Exemplo de módulo de definição



```
#ifndef EXEMP_MOD
#define EXEMP_MOD

#ifdef EXEMP_OWN
#define EXEMP_EXT
#else
#define EXEMP_EXT extern
#endif

/**** Tipo de dados exportado pelo módulo ****/
typedef struct
{
    int UmInt ;
    int OutroInt ;
} EX_tpMeuTipo ; /* declaração de tipos não é afetada pelas regras */

/**** Estruturas de dados exportada pelo módulo ****/
EXEMP_EXT int EX_vtNum[ 5 ]
#ifdef EXEMP_OWN
    = { 1 , 2 , 3 , 4 , 5 } ;
#else
    ;
#endif

#undef EXEMP_EXT
#endif
```

Padrão para o módulo de implementação



- No módulo de implementação redija o código de inclusão do respectivo módulo de definição na forma a seguir:

```
#define Nome-Arquivo-Modulo_OWN
#include "Nome-Arquivo-Modulo.H"
#undef Nome-Arquivo-Modulo_OWN
```

- Exemplo

```
/* Inclusões do compilador /
#include <stdio.h>
/* Inclusão do próprio módulo de definição /
#define EXEMP_OWN
#include "EXEMP.H"
#undef EXEMP_OWN
/* Inclusão de módulos de definição de servidores */
#include "Modulo1.H"
#include "Modulo2.H"
#include "Tabela.INC"
. . .
```

Alternativa ao padrão



- Embora fora do nosso padrão, a seguinte organização funciona também:
 - Módulo de definição (.h ou .hpp)

```
extern int XX_VarInt ; /* sempre com extern */
```
 - Módulo de implementação (.c ou .cpp)

```
int XX_VarInt = 1000 ; /* redeclarado sem o extern */
```
- Esta organização tem a **vantagem** de não necessitar os elementos **xxx_EXT**, nem a forma convoluta (enrolada) de declarar dados externados e inicializados.
- Tem a **desvantagem** de exigir a alteração de vários módulos caso seja feita alguma mudança em uma interface

Alternativa melhor ainda



- Não utilize variáveis globais externas
- Encapsule todas as variáveis globais no módulo de implementação, exemplo

```
static int ZZZ ;
```
- Acesse-as com funções de acesso declaradas no módulo de definição, exemplo

```
void AtribuirZZZ( int ZZZ ) ;  
int ObterZZZ( void ) ;
```

FIM

