Aula 06 – Ambientes de Execução

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**

Introdução

- O *front-end* do compilador realiza as análises léxicas, sintáticas e semânticas do programa fonte e constrói uma representação intermediária (IR) do código.
- O *back-end* analisa a IR e gera código-alvo, que é totalmente dependente das características da arquitetura.
- **Estes *slides***: discussão geral sobre os diferentes tipos de ambientes de execução.
- **Objetivos**: apresentar as características principais de cada ambiente.

Referências

Chapter 7 – Runtime Environments
K. C. Louden
Chapter 6 – The Procedure Abstraction
K. D. Cooper
Chapter 9 – Memory Organization
D. Thain

Introdução

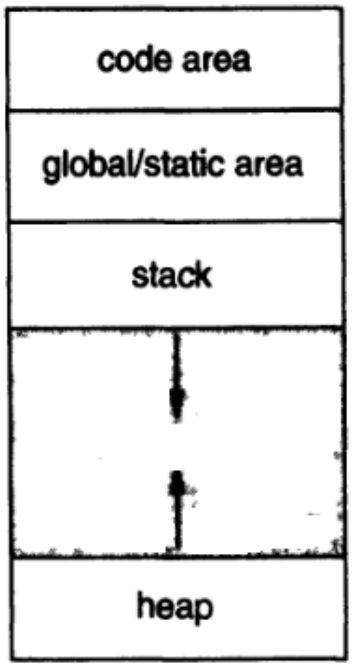
- Até agora foram estudas as fases de compilação que realizam uma análise estática da linguagem fonte.
- Isso corresponde às fases do *front-end*: análise léxica (*scanning*), análise sintática (*parsing*) e análise semântica.
- Tais análises se baseiam (dependem) somente nas propriedades da linguagem fonte.
- A partir de agora vamos estudar como um compilador gera código executável.
- Vamos ver também algumas análises adicionais realizadas para otimizar o código.
- Algumas dessas tarefas são independentes da arquitetura, mas a maior parte da tarefa de geração de código é dependente dos detalhes da máquina alvo.

Organização da Memória

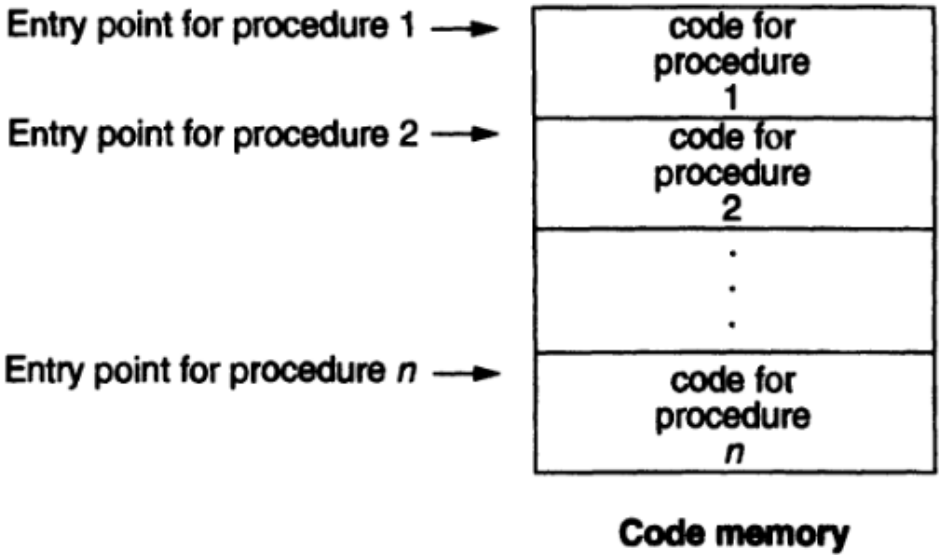
- Ambiente de execução (*runtime environment*): estrutura de registradores e de memória da máquina alvo.
- O ambiente de execução gerencia a memória e mantém as informações necessárias para guiar o processo de execução.
- Existem três tipos básicos de ambientes:
  - Totalmente estático: e.g., FORTRAN 77.
  - Baseado em pilha: e.g., C, C++, Pascal, Ada.
  - Totalmente dinâmico: e.g., LISP.
- Variações intermediárias também são possíveis.
- Algumas características das LPs determinam quais ambientes são mais adequados. Isso inclui, por exemplo:
  - Questões de escopo e alocação.
  - Natureza das chamadas de funções.
  - Mecanismos de passagem de parâmetros, etc.

Organização da Memória

A arquitetura de um computador típico é composta por um conjunto de registradores e pela memória principal (virtual), dividida em:



A área de código é fixa antes da execução e tem a seguinte organização:



Em particular, o ponto de entrada para cada função é conhecido em tempo de compilação.

Organização da Memória

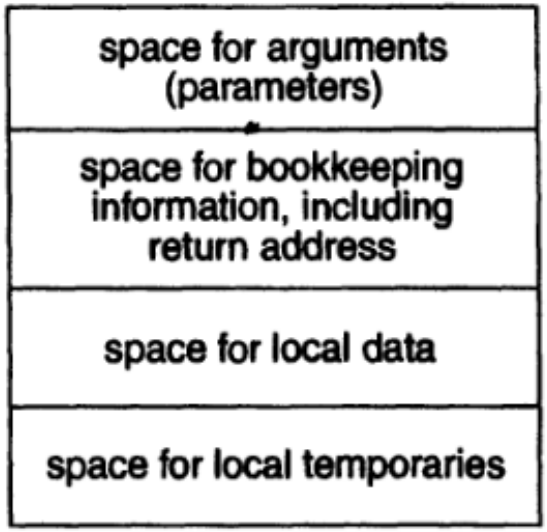
- A área de dados globais/estáticos pode ser fixada em algum endereço de memória antes da execução.
- Alocação é similar à feita para a área de código. Alguns exemplos de LPs:
  - Em FORTRAN 77, todos os dados ficam nessa área.
  - Em Pascal, as variáveis globais ficam nessa área.
  - Em C, as variáveis globais, externas e estáticas ficam nessa área.
- As constantes do código também podem ser alocadas na área de dados globais/estáticos.
- Em C isso inclui as declarações por `const` e também as *strings*.
- Convém destacar que o compilador trabalha somente com endereços virtuais de memória. O mapeamento para endereços físicos é responsabilidade do SO.

Organização da Memória

- A área de memória utilizada para armazenar dados dinâmicos pode ser organizada de diferentes formas.
- Em geral, ela é dividida em pilha e *heap*.
- A pilha é usada para dados cuja alocação segue uma política FIFO.
- O *heap* é usado para alocação explícita de áreas de memória.
- É comum que a arquitetura alvo inclua uma pilha do processador (*processor stack*) que corresponde à parte da pilha de dados apontada por um registrador especial (*stack pointer*).
- Essa pilha do processador habilita o suporte do processador para chamadas de funções e retornos.
- O compilador (montador) é responsável por providenciar a alocação explícita da pilha do processador em algum ponto apropriado da memória.

Organização da Memória

- Uma unidade importante da alocação de memória é o registro de ativação de procedimento (*procedure activation record*, ou *frame*).
- Um registro de ativação contém memória alocada para os dados locais de uma função que foi chamada.
- A organização geral de um registro de ativação é ilustrada na figura abaixo.



Organização da Memória

- Algumas partes do registro de ativação têm o mesmo tamanho para todas as funções: área de *bookkeeping*.
- Outras partes permanecem fixas para cada função: área para argumentos e dados locais.
- Algumas partes do registro de ativação podem ser alocadas e preenchidas automaticamente durante a chamada de uma função: endereço de retorno.
- Outras partes devem ser alocadas explicitamente por instruções geradas pelo compilador: espaço local de valores temporários.

Organização da Memória

- Dependendo da LP, os registros de ativação podem ser alocados em diferentes áreas da memória:
  - FORTRAN 77: na área de dados estáticos.
  - C e Pascal: na área da pilha; nessas linguagens os registros de ativação são chamados de *stack frames*.
  - LISP: na área de *heap*.
- Os registradores do processador também fazem parte da estrutura do ambiente de execução.
- Registradores podem ser usados para armazenar valores temporários, variáveis locais ou globais.
- Em arquiteturas RISC com um banco de registradores grande, os dados estáticos e até mesmo um registro de ativação inteiro pode ser armazenado nos registradores.

Organização da Memória

- Processadores possuem registradores de uso específico para manter informações da execução do programa.
  - PC – *Program Counter*: endereço da próxima instrução.
  - SP – *Stack Pointer*: endereço do topo da pilha.
  - FP – *Frame Pointer*: endereço do registro de ativação atual.
  - AP – *Argument Pointer*: endereço da área dentro do registro de ativação que armazena os argumentos.
- *Calling sequence*: sequência de operações realizadas durante uma chamada de função. Envolve as ações:
  - Alocação de memória para o registro de ativação.
  - Computação e armazenamento dos argumentos.
  - Ajustes de registradores para realizar a chamada.
- *Return sequence*: operações adicionais quando uma função retorna. Envolve as ações:
  - Cópia do valor de retorno em um local acessível ao *caller*.
  - Reajuste dos registradores.
  - Possível liberação da memória do registro de ativação.

Organização da Memória

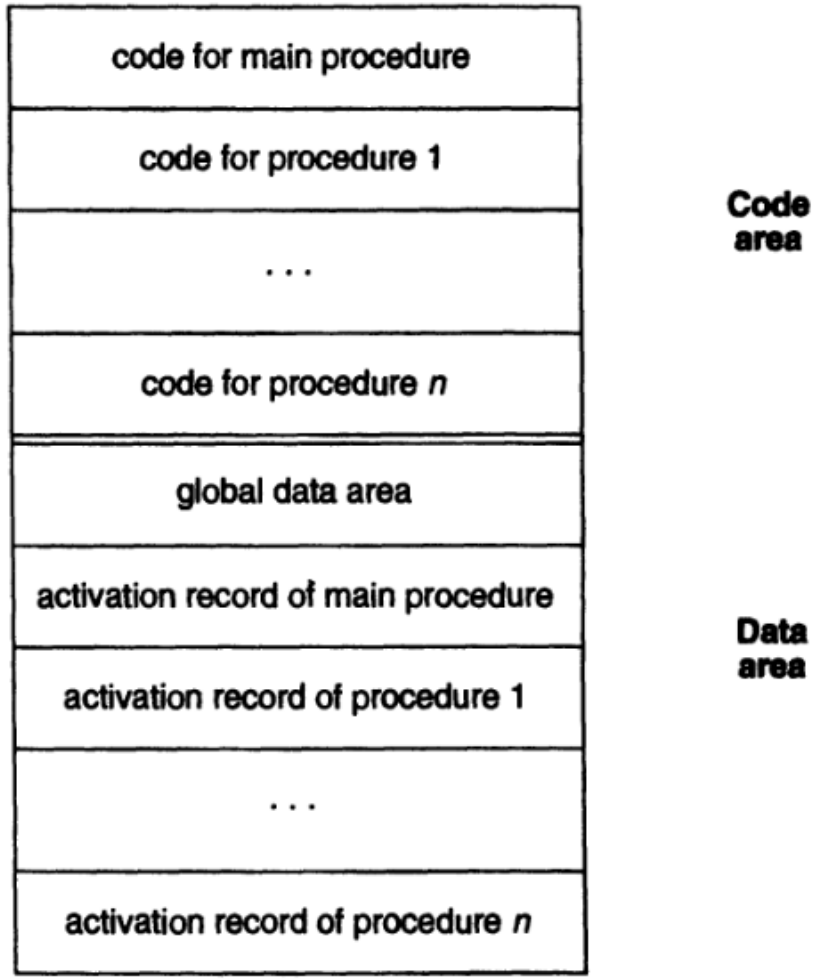
Existem dois aspectos fundamentais no projeto da *calling sequence* (sequência de ativação):

1. Como dividir as operações da sequência de ativação entre o ativador (*caller*) e o ativado (*callee*)?
   - No mínimo, o *caller* é responsável por computar os argumentos e armazená-los em localizações da memória acessíveis pelo *callee*.
2. Quais operações da sequência de ativação exigem a geração explícita de código?
   - Algumas das operações podem ter suporte nativo do processador.

Essas decisões dependem do tipo do ambiente de execução utilizado. Isso será detalhado a seguir.

Ambientes Totalmente Estáticos

Ambiente de execução mais simples de todos, aonde todos os dados são estáticos e não é permitido recursão, ponteiros ou alocação dinâmica. Memória completa do programa:



Ambientes Totalmente Estáticos

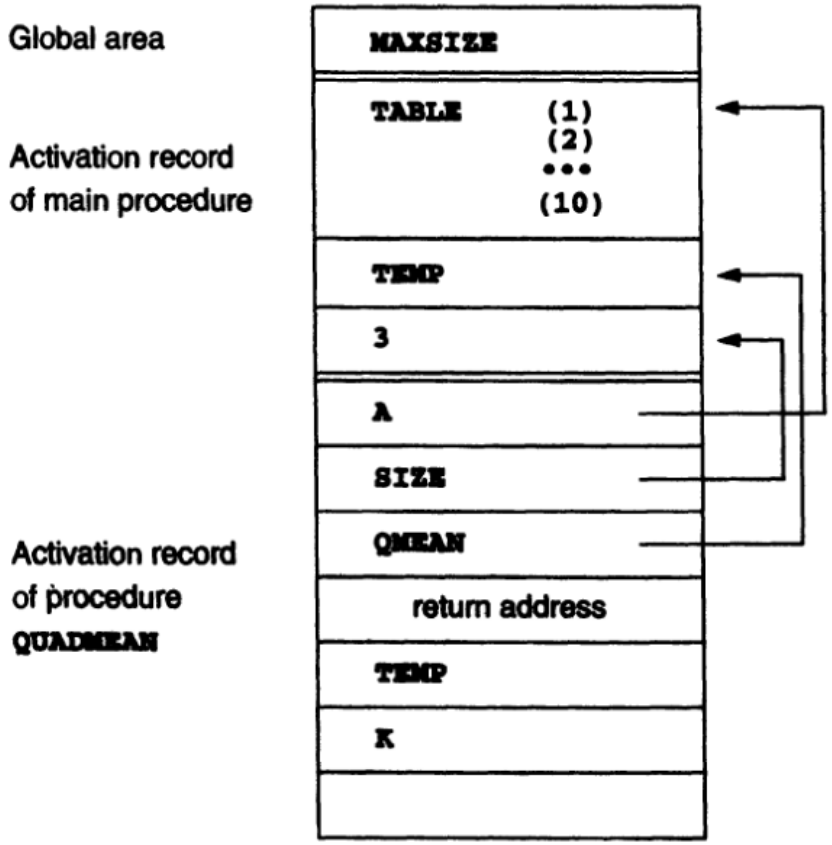
- Todos os dados são estáticos, permanecendo fixos na memória durante toda a execução do programa.
- Em uma linguagem como FORTRAN 77:
  - Todas as variáveis (globais e locais) são alocadas estaticamente.
  - Cada função tem um único registro de ativação.
  - Qualquer variável pode ser acessada diretamente em um endereço fixo.
- A sequência de ativação é simples:
  - Cada argumento é computado e armazenado na sua localização apropriada.
  - O endereço de retorno no *caller* é salvo e a execução faz um `jump` para o início do código do *callee*.
  - No término da função basta fazer outro `jump` para o endereço de retorno.

Ambientes Totalmente Estáticos

Exemplo de um programa FORTRAN 77 e seu ambiente de execução.

```
      PROGRAM TEST
      COMMON MAXSIZE
      INTEGER MAXSIZE
      REAL TABLE(10),TEMP
      MAXSIZE = 10
      READ *, TABLE(1),TABLE(2),TABLE(3)
      CALL QUADMEAN(TABLE,3,TEMP)
      PRINT *, TEMP
      END

      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      COMMON MAXSIZE
      INTEGER MAXSIZE,SIZE
      REAL A(SIZE),QMEAN, TEMP
      INTEGER K
      TEMP = 0.0
      IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
      DO 10 K = 1,SIZE
         TEMP = TEMP + A(K)*A(K)
10    CONTINUE
99    QMEAN = SQRT(TEMP/SIZE)
      RETURN
      END
```



Ambientes Totalmente Estáticos

Vantagens de um ambiente totalmente estático:

- Registros de ativação são bastante simples: única informação externa à função é o endereço de retorno.
- Causa muito pouco *overhead* com informação de *bookkeeping*.

Desvantagens/limitações de um ambiente totalmente estático:

- Não permite chamadas recursivas.
- Tamanho dos dados e sua localização na memória fixados em tempo de compilação: muito inflexível.
- Alocação dinâmica não é permitida: leva a um superdimensionamento dos dados.
- Inadequado para ambientes multi-programa e multi-usuário.

Ambientes Baseados em Pilha

- Em uma LP que permite chamadas recursivas as variáveis locais devem ser alocadas novamente a cada chamada.
- Isso implica que registros de ativação não podem ser alocados de forma estática.
- A alternativa mais comum é alocar os registros de ativação em uma pilha, cujo tamanho varia conforme a sequência de chamadas do programa.
- Uma mesma função pode ter diferentes registros de ativação na pilha, cada um representando uma chamada distinta.
- Requer uma estratégia mais complexa de *bookkeeping* e acesso às variáveis.
- Essa estratégia é totalmente dependente das características da LP fonte.

Ambientes de Pilha sem Procedimentos Locais

Em uma linguagem aonde todas as funções são globais (i.e., não podem ser aninhadas), como em C, o ambiente baseado em pilha exige:

1. Um *frame pointer* (fp): ponteiro para o registro de ativação atual para permitir o acesso às variáveis locais.
2. Um *control (dynamic) link*: ponteiro para o registro de ativação da função anterior.
3. Um *stack pointer* (sp): ponteiro para o topo da pilha.

Ambientes de Pilha sem Procedimentos Locais

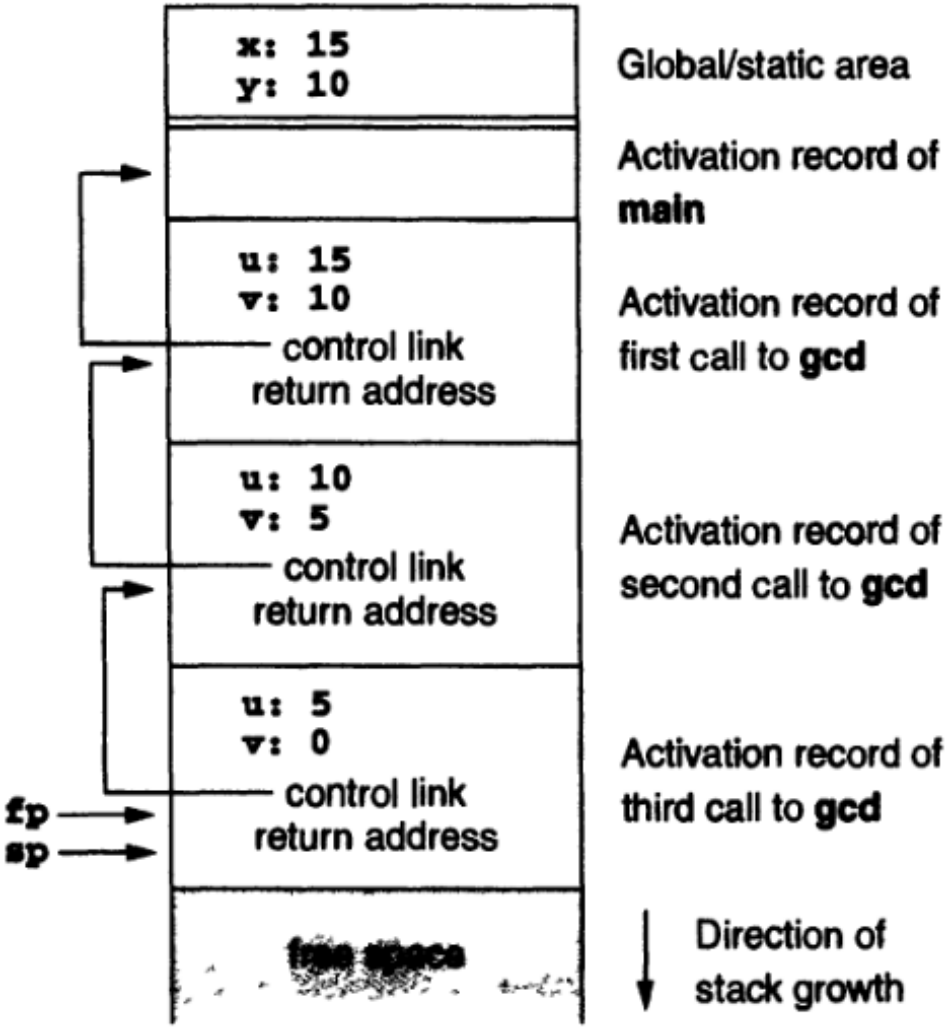
Suponha que o algoritmo de Euclides abaixo recebe 15 e 10 como entrada. Estado da pilha durante a terceira chamada:

```
#include <stdio.h>

int x, y;

int gcd(int u, int v) {
  if (v == 0) return u;
  else return gcd(v, u%v);
}

int main() {
  scanf("%d%d", &x, &y);
  printf("%d\n", gcd(x,y));
  return 0;
}
```



Ambientes de Pilha sem Procedimentos Locais

Acesso aos nomes:

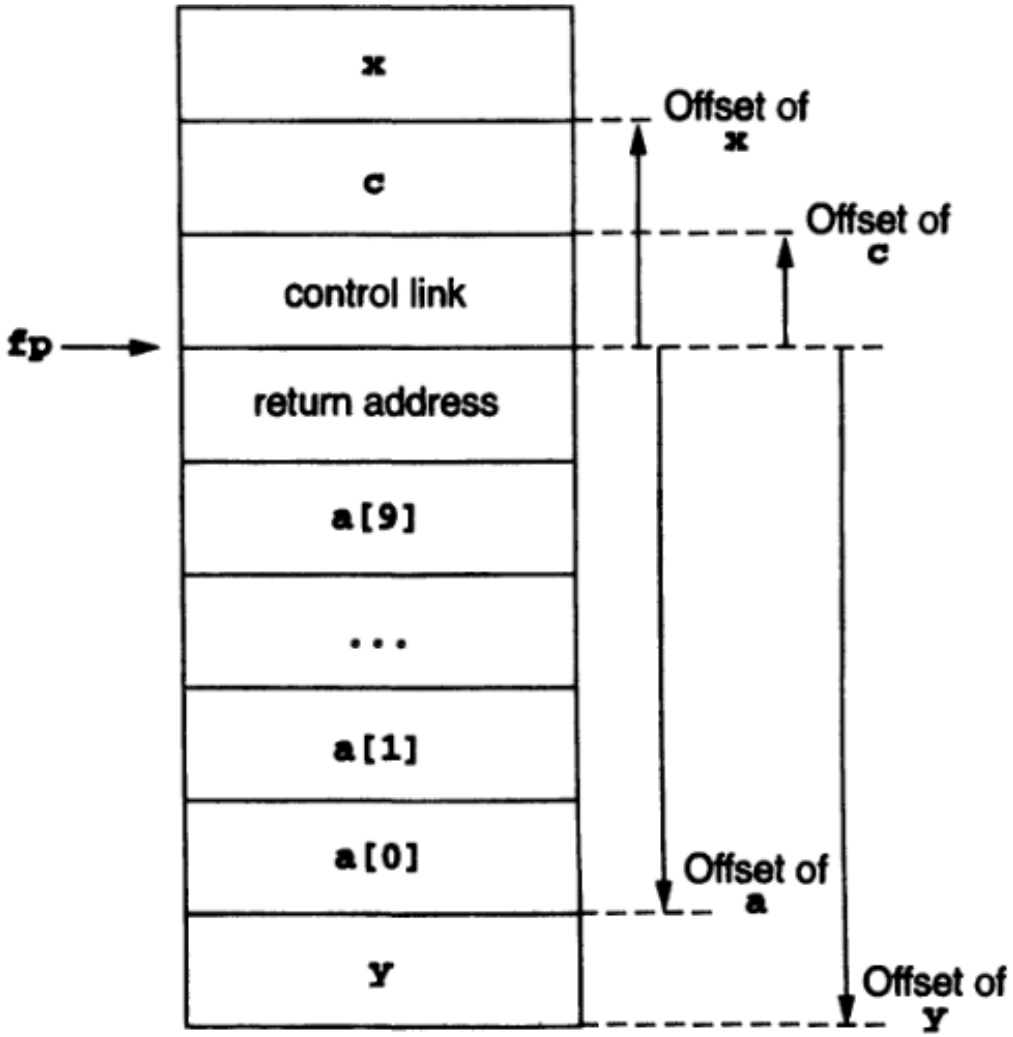
- Parâmetros e variáveis locais não podem mais ser acessados em um endereço fixo.
- Para cada nome, é necessário computar um *offset* a partir do *frame pointer* atual.
- Na maioria das LPs, esses *offsets* podem ser computados estaticamente pelo compilador.

Ambientes de Pilha sem Procedimentos Locais

Considere a função e registro de ativação abaixo. Parâmetros e variáveis locais têm *offsets* com sinais opostos devido à posição relativa ao *fp*.

```
void f(int x, char c) {
  int a[10];
  double y;
  ...
}
// int = 2 bytes
// addr = 4 bytes
// char = 1 byte
// double = 8 bytes
// a[i] = (-24+2*i)(fp)
```

Name	Offset
x	+5
c	+4
a	−24
y	−32



Ambientes de Pilha sem Procedimentos Locais

Quando uma função é chamada, devemos:

- Computar os argumentos e armazená-los em suas respectivas posições no novo *frame* da função.
- Armazenar (*push*) o **fp** como o *control link* do novo *frame*.
- Atualizar o **fp** para o início do novo *frame*: **fp** $\leftarrow$ **sp**.
- Armazenar o endereço de retorno no novo *frame*.
- Fazer um `jump` para o código da função chamada.

Quando uma função termina, devemos:

- Copiar **fp** para **sp**.
- Carregar o *control link* (acessível em `+4(sp)`) em **fp**.
- Se a função tiver valor de retorno, acessar a sua posição a partir do novo **sp** e copiar para um local temporário.
- Fazer um `jump` para o endereço de retorno (em `-4(sp)`).

Ambientes de Pilha sem Procedimentos Locais

Lidando com dados de tamanhos variados:

- Algumas funções aceitam um número variado de argumentos (e.g, funções *vararg* em C).
- O compilador C lida com funções *vararg* empilhando os argumentos em ordem inversa.
- Isso garante que o primeiro parâmetro fique sempre localizado em um *offset* fixo do **fp**.
- Outra possibilidade: em algumas arquiteturas (e.g., VAX) existe um *argument pointer (ap)* para indicar a localização dos argumentos.
- Pouco comum atualmente.

Ambientes de Pilha sem Procedimentos Locais

Lidando com valores temporários locais:

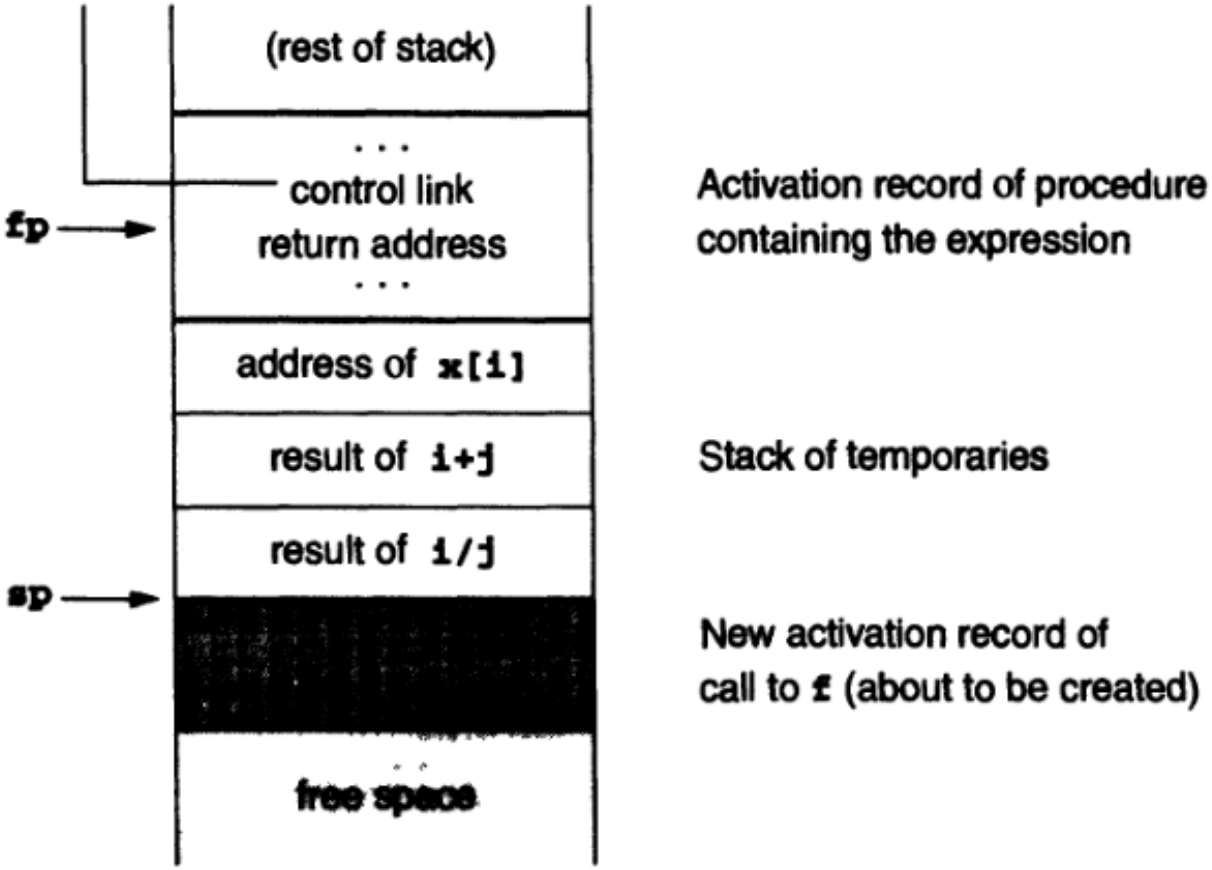
- Valores temporários locais são resultados parciais de uma computação que devem ser armazenados em cada registro de ativação.
- Considere, por exemplo, a expressão em C:

```
x[i] = (i + j) * (i/k + f(j))
```

- Em uma avaliação da esquerda para a direita da expressão, três valores temporários são calculados antes da chamada de `f(j)`.
  - O endereço de `x[i]`, para a atribuição pendente.
  - A soma `i+j`, para a multiplicação pendente.
  - O quociente `i/k`, para a soma pendente.
- Esses valores temporários podem ser armazenados no topo da pilha, enquanto forem necessários. (Também podem ficar em registradores.)

Ambientes de Pilha sem Procedimentos Locais

Configuração da pilha no exemplo do slide anterior antes da chamada da função.



Outros Ambientes Baseados em Pilha

- Existem outros tipos de ambientes baseados em pilha mais complexos.
- Complexidade surge pelo aninhamento de funções/procedimentos.
- Esses ambientes não serão estudados aqui.
- Para mais informações, veja as seções 7.3.2 e 7.3.3 do livro do Louden.

Ambientes de Execução Totalmente Dinâmicos

- Em ambientes de execução baseados em pilhas podem surgir referências pendentes (*dangling references*).
- Uma pendência pode surgir quando uma referência a uma variável local de uma função for retornada, como no exemplo abaixo em C:

```
int* dangle(void) {
  int x;
  return &x;
}
```

- Uma atribuição como `addr = dangle()` faz `addr` apontar para um endereço inválido da pilha de ativação.

Ambientes de Execução Totalmente Dinâmicos

- Um caso mais complexo de referência pendente ocorre se uma função local puder ser retornada.
- A linguagem C evita esse problema proibindo funções aninhadas.
- Outras linguagens como Modula-2, Ada, etc, permitem aninhamento de funções e precisam de uma análise mais detalhada para evitar referências pendentes.
- Em linguagens funcionais, as funções podem ser definidas localmente, passadas como parâmetros e retornadas por outras funções.
- $\Rightarrow$ Um ambiente baseado em pilha não é adequado para esse tipo de LP.

Ambientes de Execução Totalmente Dinâmicos

- A alternativa é usar um ambiente totalmente dinâmico.
- Nesse tipo de ambiente, a estrutura básica do registro de ativação permanece a mesma, no entanto, a sua política de desalocação é diferente.
- Quando uma função termina e o controle de execução volta para o *caller*, o *frame* do *callee* permanece alocado na memória, para evitar referências pendentes.
- A desalocação ocorre somente quando for segura, i.e., quando não houver mais referências para o *frame*.
- Como o controle de desalocação é mais complexo, LPs com esse tipo de ambiente geralmente usam um coletor de lixo.

Memória Dinâmica em Linguagens OO

- Uma LP orientada a objetos (OO) exige mecanismos especiais do ambiente de execução para permitir a implementação das suas características adicionais.
- Exemplos dessas características incluem: objetos, métodos, herança e despacho dinâmico de métodos.
- Nível de exigência sobre o ambiente de execução varia bastante dependendo da LP.
- Exemplo dos extremos:
  - A linguagem Smalltalk requer um ambiente totalmente dinâmico.
  - A linguagem C++ mantém o ambiente de pilha do C o máximo possível.
- Objetos na memória podem ser vistos como um cruzamento entre uma estrutura de dados tradicional e um registro de ativação.

Memória Dinâmica em Linguagens OO

Objetos (instâncias de classes) podem ser implementados de diferentes formas:

1. Área de memória do objeto passa por um código de inicialização que copia todos os membros (campos e métodos) herdados diretamente para a estrutura do objeto. Métodos são armazenados como ponteiros para código.
2. A estrutura completa das classes é mantida em memória através de um grafo de heranças. Objetos mantém ponteiros para a sua classe.
3. Computar uma lista de ponteiros de código para os métodos disponíveis em cada classe e armazenar a lista em memória como uma tabela de funções virtuais.

Exemplos de LPs: 1 – Python; 2 – Smalltalk; 3 – C++.

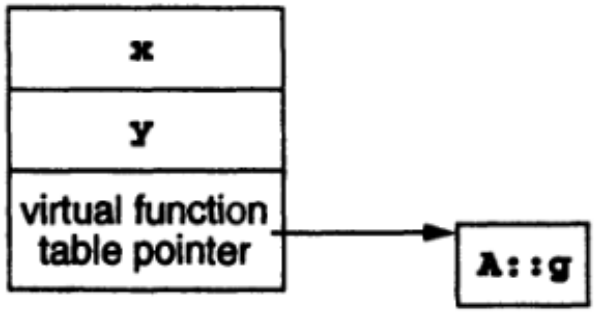
Memória Dinâmica em Linguagens OO

Exemplo em C++.

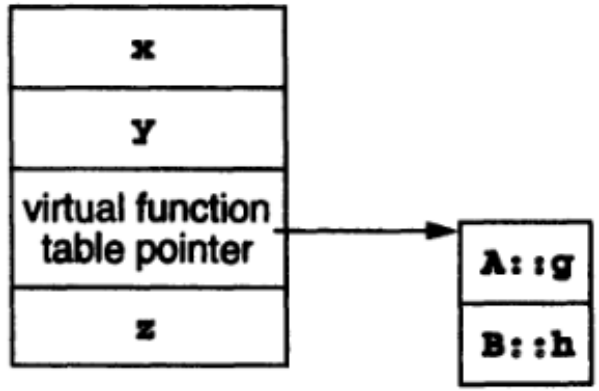
```
class A {
  public:
    double x, y;
    void f();
    virtual void g();
};

class B: public A {
  public:
    double z;
    void f();
    virtual void h();
};
```

Objetos da classe A:



Objetos da classe B:



Função `f()` não aparece na tabela por não ser `virtual`.

Gerenciamento do *Heap*

- A maioria das LPs, mesmo as com ambientes baseados em pilha, precisa de alguma funcionalidade de alocação de dinâmica de memória.
- *Heap*: área de memória que cuida desse tipo de alocação.
- Operações básicas do *heap* envolvem alocação e/ou desalocação de blocos contínuos de memória.
- Funcionamento/controle dessa alocação depende das definições da LP.
- Alocação geralmente é explícita. Exceção: LPs funcionais.
- Desalocação pode ser explícita (C, C++) ou implícita (Java, Python).
- Desalocação implícita exige coleta de lixo (*GC – garbage collection*).
- Algoritmo mais comum para GC: *mark-and-sweep*.

Mecanismos de Passagem de Parâmetros

- Em uma chamada de função, os parâmetros são posições do *frame* que devem ser preenchidos pelo ativador com os argumentos.
- Esse processo é chamado amarração entre parâmetros e argumentos.
- Mecanismos de amarração (passagem) de parâmetros variam por LP. Tipos principais:
  - Passagem por valor.
  - Passagem por referência.
  - Passagem por valor-resultado.
  - Passagem por nome.
- Outro detalhe que precisa ser definido: ordem de avaliação dos argumentos.
- Alguns compiladores C mais antigos avaliam os argumentos da direita para a esquerda, o que pode causar confusão em chamadas como `f(++x, x);`.

Passagem por Valor

- Argumentos são expressões que são avaliadas no momento da chamada da função.
- Os valores das expressões passam para os parâmetros durante a execução da função.
- Este é o único tipo de mecanismo em C.
- Em outras LPs como Pascal e Ada esse é o mecanismo padrão (que pode ser modificado).
- Método mais simples, não requer esforço adicional por parte do compilador.

Passagem por Referência

- Mecanismo passa a localização (endereço) do argumento, de forma que o parâmetro é um *alias* para o argumento.
- Único mecanismo de passagem de parâmetro em FORTRAN 77.
- Em Java é o padrão para objetos.
- Em C++ precisa ser especificado pelo símbolo &.
- Esse mecanismo requer que o compilador compute o endereço do argumento para armazená-lo no registro de ativação.
- Complicações:
  - Acesso local (dentro da função) ao parâmetro levam a acessos indiretos, pois o valor está em um endereço fora do *frame* atual.
  - Em chamadas como `p(x+3)`, o compilador precisa alocar um endereço temporário para o resultado da expressão `x+3` antes de fazer a chamada para `p`.

Passagem por Valor-Resultado e por Nome

Passagem por valor-resultado:

- Similar à passagem por referência mas sem *alias*.
  Também conhecido como *copy-restore*.
- Implementado em Ada como parâmetros do tipo `in out`.

Passagem por nome:

- Mecanismo mais complexo de passagem de parâmetros.
  Também conhecido como avaliação tardia (*delayed*).
- Nesse mecanismo o argumento não é avaliado até o seu uso na função chamada.
- Mecanismo introduzido na LP Algol 60.
- Não é mais usado em LPs imperativas atuais por tornar o código difícil de ler, além de ser ineficiente.
- Uma variação desse mecanismo chamado *lazy evaluation* virou o mecanismo padrão em LPs funcionais.

Aula 06 – Ambientes de Execução

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**