Universidade Federal de Campina Grande
Departamento de Sistemas e Computação
Curso: Ciência da Computação
Disciplina: Paradigmas de Linguagens de Programação
Professor: Franklin Ramalho
Período: 2008.1
Grupo: Ana Gabrielle Ramos Falcão
Fabiano de Miranda Silva
Paulo Ricardo Motta Gomes
Renato Miceli Costa Ribeiro

Linguagem C

Campina Grande, 29 de julho de 2008

Índice Analítico

Índice Analítico....2
Índice de Figuras....3
1.A linguagem C....6
1.1.História....6
1.2.Características....6
1.3.Sintaxe e Semântica....7
2.Valores e tipos....7
2.1.Tipos Primitivos....8
2.2.Tipos Compostos....8
2.2.1.Produto Cartesiano....8
2.2.2.União disjunta....9
2.2.3.Mapeamento....9
2.2.4.Conjuntos Potência....11
2.2.5.Tipos Recursivos....11
2.2.6.Strings....11
2.3.Sistema de Tipos....12
2.4.Classificação de Valores....12
3.Expressões....13
3.1.Expressões Agregadas....13
3.2.Expressões Condicionais....14
4.Comandos....15
4.1.Comandos Condicionais....15
5.Variáveis....16
5.1.Variáveis Armazenáveis....16
5.2.Atualização....16
5.3.Arrays....17
5.4.Variáveis Heap (Apontadores)....17
5.5.Variáveis Persistentes....19
6.Associações....20
6.1.Entidades Associáveis....20
6.2.Blocos....21
6.3.Estrutura de blocos....22
6.4.Escopo....23
6.5.Definições....23
6.5.1.Declaração e definição de tipos....24
6.5.2.Declaração de variáveis....24
6.5.3.Declarações seqüenciais....25
6.5.4.Declarações colaterais....25
6.5.5.Declarações recursivas....25
6.5.6.Comandos-Bloco....26
6.5.7.Expressões Bloco....26
7.Abstrações....26
8.Passagem de parâmetros....27
8.1.Passagem de parâmetros por cópia....28
8.2.Passagem de parâmetros por definição....28
9.Encapsulamento....28
9.1.1.Pacotes....28

9.1.2.TADs....30
9.1.3.Objetos....30
10.Sistema de tipos....30
10.1.Sobrecarga....30
10.2.Polimorfismo....30
10.3.Coerção....31
10.4.Subtipos e herança....31
11.Seqüenciadores....31
11.1.Desvio incondicional....31
11.2.Escape....32
11.3.Exceções....33
12.Concorrência....33
12.1.Regiões Críticas....34
12.1.1.Variáveis mutex....34
12.1.2.Spinlocks....36
12.1.3.Semáforos....37
12.2.Eventos....39
12.3.Mensagens....40
12.4.RPC 41
13.Decisões de Projeto....43
14.Conclusão....43
15.Referências....44

Índice de Figuras

Figura 1 - Exemplo de código na linguagem C.....7
Figura 2 - Exemplo de produto cartesiano na linguagem C.....8
Figura 3 - Múltiplas formas de se declarar registros.....8
Figura 4 - União livre (insegura) em C.....9
Figura 5 - Simulação de união disjunta em C.....9
Figura 6 - Declaração de mapeamento array.....10
Figura 7 – Inicialização de mapeamento array em C.....10
Figura 8 – Mapeamentos em C.....10
Figura 9 - Simulação de tipo recursivo através de produto cartesiano e apontador.....11
Figura 10 – Uso de string como array de caracteres.....11
Figura 11 – Exemplo de checagem fraca de tipos.....12
Figura 12 – Parametrização de array via apontador.....12
Figura 13 – Expressão agregada sobre produto cartesiano.....13
Figura 14 – Expressões agregadas em C.....13
Figura 15 – Uso de expressão condicional em C.....14
Figura 16 – Comando condicional if.....15
Figura 17 – Comando condicional switch.....15
Figura 18 – Atualizações total e seletiva sobre registros.....16
Figura 19 – Atualizações total e seletiva sobre arrays.....17
Figura 20 – Instanciação de array estático.....17
Figura 21 – Simulação de array dinâmico.....17
Figura 22 – Declaração e atribuição de apontador para inteiro.....18
Figura 23 – Alocação de espaço em memória para apontadores.....18
Figura 24 – Desalocação de espaço em memória para apontadores.....19
Figura 25 – Variáveis persistentes em C.....20
Figura 26 – Diferentes ambientes em um programa C.....20
Figura 27 – Associações entre entidades e identificadores.....21
Figura 28 – Bloco principal “main”.....21
Figura 29 – Aninhamento de blocos.....22
Figura 30 – Escopo.....23
Figura 31 – Definição de constante.....23
Figura 32 – Declaração de um novo tipo.....24
Figura 33 – Definição de tipo.....24
Figura 34 – Declaração de variáveis.....25
Figura 35 – Declaração de variável utilizando tipo definido por usuário.....25
Figura 36 – Declaração seqüencial.....25
Figura 37 – Declaração recursiva.....25
Figura 38 – Comando-bloco.....26
Figura 39 – Expressão-bloco.....26
Figura 40 – Estrutura de uma abstração de função.....27
Figura 41 – Exemplo de abstração de procedimento.....27
Figura 42 – Passagem de parâmetro por cópia.....28
Figura 43 – Arquivo de cabeçalho matematica.h.....29
Figura 44 – Arquivo de código matematica.c.....29
Figura 45 – Arquivo que inclui pacote matematica.h 29
Figura 46 – Arquivo que tenta acessar elementos inacessíveis do pacote.....30
Figura 47 – Coerção em C.....31

Figura 48 – Subtipos primitivos.....31
Figura 49 – Desvio incondicional.....32
Figura 50 – Escape : return.....32
Figura 51 – Escape : exit().....32
Figura 52 – Escape : break.....33
Figura 53 – Escape : continue.....33
Figura 54 – Utilização de mutex.....35
Figura 55 – Utilização de spin-lock.....37
Figura 56 – Utilizando mutex através de semáforo binário.....39
Figura 57 – Exemplo da utilização de eventos.....40
Figura 58 – Envio e recebimento de mensagens com MPI.....41
Figura 59 – Stub RPC servidor - rpcserver.c.....42
Figura 60 – Stub RPC cliente - rpcclient.c.....43

1. A linguagem C

1.1. História

O desenvolvimento inicial da Linguagem de Programação C ocorreu no AT&T Bell Labs entre 1969 - 1973 por Dennis Ritchie e Ken Thompson, baseando-se em muitos dos princípios da Linguagem B (previamente escrita por Thompson). Foi originalmente criada para a implementação do sistema operacional UNIX, sendo atualmente uma das linguagens mais usadas por todo o mundo.

Em conseqüência de seu poder e flexibilidade, o sistema operacional UNIX (que havia sido originalmente implementado em Assembly) foi quase completamente reescrito em C. Pelo resto dos anos 1970, esta linguagem de programação se tornou cada vez mais popular e se difundiu pelas universidades por causa de seus laços com UNIX e a disponibilidade de seus compiladores.

Nos anos seguintes, sua difusão tornou-se mais intensa, e a diversidade de variantes gerou problemas de compatibilidade. Em 1983, a American National Standards Institute (ANSI) formou um comitê com o propósito de estabelecer uma definição padrão de C - ANSI C, que foi ratificada em 1989. Em 1990, o padrão ANSI foi reformado e adotado pela International Organization for Standardization (ISO). O padrão ANSI/ISO foi reformulado novamente em 1999, e ficou conhecido por C99.

A linguagem C ainda permanece bastante utilizada atualmente. Influenciou a definição de diversas outras linguagens de programação, como D, Java e Perl, além de suas sucessoras diretas C++ e C#. Ainda, algumas linguagens interpretadas, como Python, possuem interpretadores escritos puramente em linguagem C.

1.2. Características

C é uma linguagem poderosa e flexível, que permite grande liberdade ao programador. A linguagem é traduzida para código de máquina e impõe poucas restrições quanto ao manuseio de uso do processador e acesso à memória. Essas qualidades fazem dela uma linguagem útil para programação tanto de softwares simples quanto de sistemas de larga escala.

A velocidade com que programas C são executados se deve à compilação gerar diretamente código de máquina. Sua flexibilidade decorre da possibilidade de codificar vários algoritmos que realizem a mesma tarefa. C possui operadores de bit-a-bit e poderosa capacidade de manipulação de ponteiros. A linguagem impõe poucas restrições ao programador, dada a possibilidade de coerção direta entre os vários tipos disponíveis.

Um ponto forte da linguagem C é o uso de modularidade. O código pode ser armazenado em pacotes, denominados bibliotecas, para reuso em programas futuros. Esse conceito de modularidade também contribui para a portabilidade e rápida codificação de programas em C. A linguagem dispõe de uma biblioteca padrão, chamada “libc”, que dispõe diversas funções para uso em seus programas.

A linguagem C é considerada confiável, uma vez que sempre há a garantia de que uma codificação feita na linguagem vai executar exatamente o que foi definido por sua semântica. Para exemplificar isso, podemos considerar o caso dos sistemas operacionais: SOs são softwares complexos e que realizam as mais diversas tarefas – e para tanto precisam ser confiáveis –, e todos os grandes exemplos de SOs foram codificados em C. Ainda, a linguagem é considerada legível, por possuir uma sintaxe bem definida que faz distinção entre definições, declarações e atribuições, e, em cada

um desses, entre variáveis, comandos e expressões. Por fim, ela também é considerada segura: programas C podem ser projetados para manter a informação confidencial, integral e disponível. Podemos afirmar que os próprios programas C podem ser codificados de modo a serem seguros perante a execução de outros softwares no mesmo sistema.

A linguagem C é uma das representantes mais populares do paradigma de programação imperativo. Este paradigma recebe este nome por ser baseado em comandos que atualizam variáveis armazenadas, ou seja, que modificam o estado de um programa. Mais especificamente, a linguagem C pode ser classificada como integrante do paradigma procedural ou procedimental, que se baseia no conceito de chamada de subprogramas, ou procedimentos. Além disso, C é uma linguagem bloco-estruturada, dado que fornece estruturas hierárquicas que possuem início e fim (blocos), e geralmente são úteis na modularização interna de um programa (como laços, execuções condicionais e sub-rotinas). Apesar de a programação estruturada não permitir o uso de comandos incondicionais (goto), C disponibiliza essa funcionalidade, porém o seu uso é restrito, senão desaconselhado.

A figura 1 mostra um exemplo simples de código em C, onde é possível observar o aspecto procedural e bloco-estruturado da linguagem presentes na rotina principal main().

```
1  #include <stdio.h>
2
3  int main() {
4     printf("Hello World!\n");
5     return 0;
6  }
```

Figura 1 - *Exemplo de código na linguagem C*

C é uma linguagem compilada, ou seja, o seu código tem que ser compilado para linguagem de máquina para ser executado. Existem diversos compiladores para vários padrões C, porém, a maioria é compatível com o padrão ANSI C. Dentre os vários compiladores, podemos destacar os mais usados: GNU Compiler Collection (gcc), o Intel C/C++ Compiler (icc), a Sun Studio, a Borland C/C++ Compiler e a Microsoft Visual C/C++ Compiler.

1.3. Sintaxe e Semântica

A sintaxe de C é definida através de uma gramática na Forma de Backus-Naur, como especificado por Kernighan e Ritchie em [37].

Sua semântica, como bastante pesquisada por Nikolaos S. Papaspyrou (trabalhos em [38]) é denotacional.

2. Valores e tipos

Dos valores encontrados em C, podemos citar como exemplos os numéricos (**doubles**, **ints**, **floats** e **enums**), os simbólicos (**chars**, strings), os referenciais (ponteiros) e os valores compostos (**unions**, **structs** e arrays).

2.1. Tipos Primitivos

A linguagem C possui tipos primitivos predefinidos, como **char** (Character), **int** (Integer), **float** (Float) e **double** (Double). A esses tipos podem ser aplicados modificadores, como **long** e **short** para **int**, e **signed** e **unsigned** para todos os tipos. O tamanho de cada tipo depende dos modificadores que lhe são aplicados e da arquitetura da máquina. O intervalo de valores que eles suportam também depende dos modificadores e da arquitetura, sendo menor na aplicação de **short** e maior na aplicação de **long**.

Em C, não existe um tipo booleano. Os valores booleanos podem ser considerados uma redefinição dos valores inteiros: 0 é considerado *false*, enquanto qualquer outro valor é *true*.

2.2. Tipos Compostos

2.2.1. Produto Cartesiano

O produto cartesiano é disponibilizado em C na forma de um registro, representado através da palavra-chave **struct**. **Struct** é utilizado para declarar um novo agrupamento de variáveis, que não possuem entre si necessariamente nenhum relacionamento. Na figura 2 podemos ver um exemplo de produto cartesiano de nome *_Campo*; variáveis do tipo *_Campo* possuem três campos: um apontador para apontador de Casa (matriz de Casa), acessado através do identificador *tabuleiro*, e dois inteiros de nomes *totBombas* e *tamanho*.

```
1  struct _Campo {
2       Casa *(*tabuleiro);
3       int totBombas;
4       int tamanho;
5  };
```

Figura 2 - ***Exemplo de produto cartesiano na linguagem C***

Produtos cartesianos podem ser declarados de diversas formas, e não necessariamente estarem atrelados a um identificador de tipo.

```
1  struct Xpto {int a; int b; char c;};
2
3  struct Xpto variable;
4
5  struct {int a; int b; char c;} variable;
```

Figura 3 - ***Múltiplas formas de se declarar registros***

Os exemplos das linhas 1 e 5 da figura 3 determinam declarações de produtos cartesianos. Nas linhas 3 e 5 da mesma figura, uma variável é associada ao tipo definido pelo produto cartesiano, cujo identificador é determinado *variable*. Nas linhas 1 e 3 da figura 3, o produto cartesiano é identificado por **Xpto**; na linha 5, o produto cartesiano definido não possui um identificador estabelecido: a única variável que armazena valores desse tipo é *variable*. Todos os campos do produto cartesiano são acessíveis.

2.2.2. União disjunta

A linguagem C não suporta diretamente a união disjunta. Apesar de C possuir o tipo **union**, que representa uniões livres, ele não é considerado uma união disjunta por não possuir um rótulo (*tag*) anexado. Sem um rótulo é impossível realizar um teste para saber a que conjunto o elemento pertence, e isso faz com que a recuperação do elemento (a projeção) seja insegura.

```
1  union numero_sem_tag {
2     int valorInt;
3     float valorFloat;
4  };
```

Figura 4 - ***União livre (insegura) em C***

A **union** é declarada da mesma forma de uma **struct;** porém, ao invés de alocar uma região na memória para cada elemento, aloca apenas uma região para toda a estrutura, do tamanho do seu maior elemento. Se for alocada uma variável do tipo **numero_sem_tag**, cuja união é definida na figura 4, e um número for armazenado no campo *valorFloat* e lido através do campo *valorInt*, haverá uma inconsistência de dados, o que não deve ocorrer na união disjunta. Para simular uma união disjunta em C, envolvemos uma **union** em um **struct** com um rótulo, que identifica de que tipo será o valor armazenado, como mostra a Figura 5. Essa simulação, entretanto, exige cautela, pois ainda será necessário realizar o teste com o rótulo para identificar a qual dos conjuntos da união disjunta pertence o valor e utilizar o campo adequado para realizar a leitura desse valor.

```
1  enum Precisao {exato, inexato};
2
3  struct Numero {
4     Precisao tag;
5     union {
6        int valorInt;       // usado quando tag contem exato
7        float valorFloat;   // usado quando tag contem inexato
8     } conteudo;
9  };
```

Figura 5 - ***Simulação de união disjunta em C***

2.2.3. Mapeamento

A funcionalidade de mapeamento em C é representada pelos arrays e pelas abstrações de função.

Array é um tipo de estrutura de dados que comporta múltiplas variáveis do mesmo tipo e mapeia um inteiro neste tipo. Os índices dos elementos de um array são valores de 0 até seu tamanho decrescido de uma unidade. Os arrays devem ser declarados especificando-se seu tipo e seu tamanho, no seguinte formato:

```
tipoDeDado nomeDoArray[tamanhoDoArray]
```

```
1  int leInteiro(char *mensagem, int limiteInferior, int
2                limiteSuperior, char *mensagemErro) {
3
```

```
4       char input[4];
5       int result;
6
7       (...)
8
9   }
```

Figura 6 - ***Declaração de mapeamento array***

Na linha 4 da figura 6 vemos uma declaração de mapeamento array em C. A variável de identificador *input* mapeia quatro valores de **int** em **char**, cujos elementos são acessados através dos índices 0 a 3.

Arrays também podem ser inicializados no momento da sua declaração, através do uso de expressões agregadas (que serão detalhadas na seção 3.1). O formato de inicialização durante a declaração deve ser como se segue:

```
tipoDeDado nomeDoArray[tamanhoDoArray] = {listaDeValores}
```

```
1   char caracteres[5] = {'a', 'b', 'c', 'd', 'e'};
```

Figura 7 – Inicialização de m*apeamento array em C*

Na linha 1 da figura 7, vemos a inicialização da variável *caracteres* (mapeamento array de 5 posições de **char**) com sua lista de valores. Esses valores são ordenados da esquerda para a direita, com cada valor associado a um **int** diferente, uma unidade maior que o **int** associado ao valor à esquerda (o primeiro valor será associado a zero). Todos os valores na lista de valores devem estar separados por vírgula.

Arrays são acessados através de seu identificador, seguido da posição desejada entre colchetes, como pode ser visto abaixo:

```
nomeDoArray[indiceDoArray]
```

Além de arrays, mapeamentos podem ser implementados através de abstrações de função. Uma abstração de função mapeia o produto cartesiano de seus argumentos em seu tipo de retorno. O valor mapeado não necessariamente deve corresponder ao valor fornecido à entrada da abstração de função.

Para acessar o valor mapeado por uma abstração de função, basta realizar a chamada da abstração, passando como argumentos os valores que deseja mapear para o tipo de retorno.

```
1   Casa * getCasa(Campo campo, Pos pos) {
2         return &campo.tabuleiro[pos.m + 1][pos.n + 1];
3   }
4
5   void setCasa(Campo campo, Pos pos, Casa casa) {
6         campo.tabuleiro[pos.m + 1][pos.n + 1] = casa;
7   }
8
9   int ehBomba(TipoCasa tipo) {
10        return ((tipo >= bomba) && (tipo <= bombaComputador));
11  }
```

Figura 8 – ***Mapeamentos em C***

O acesso a um valor mapeado por um array é feito como exemplificado na linha 2 da figura 8. Na linha 6 da mesma figura, podemos ver uma atualização seletiva da única posição do array com o valor *casa*. Ainda, nas linhas 1-3 e 9-11 da figura 8, vemos mapeamentos do tipo abstração de função, em que os tipos **Campo** x **Pos** e **TipoCasa** são mapeados para **Casa*** e **int**, respectivamente.

2.2.4. Conjuntos Potência

C não oferece conjuntos potência como tipo nativo.

2.2.5. Tipos Recursivos

A linguagem C não suporta tipos recursivos diretamente. Entretanto, é possível simular tipos recursivos através do uso de produtos cartesianos e apontadores.

```
1  struct Nodo {
2     int elem;
3     struct Nodo * prox;
4  };
```

Figura 9 - ***Simulação de tipo recursivo através de produto cartesiano e apontador***

No caso mostrado na figura 7, a estrutura **Nodo** é declarada de forma recursiva. Neste caso, um valor de **Nodo** representa um nodo de uma lista encadeada simples. Cada nodo da lista contém um inteiro *elem* e um apontador *prox* para o próximo nodo, com exceção do último nodo, cujo valor será um ponteiro nulo (tipicamente representado por 0).

2.2.6. Strings

C trata Strings como um tipo composto. Elas são arrays de **char**, onde cada caractere da string ocupa uma posição do array. Convencionou-se que o caracter "null" (\0) é armazenado na primeira posição após o final da palavra marcar o fim da string.

```
1  char input[4];
2  int result;
3
4  for (;;) {
5     printf(mensagem);
6     printf(" (%d-%d) ", limiteInferior, limiteSuperior);
7     scanf("%s", input);
8
9     (...)
10
11 }
```

Figura 10 – ***Uso de string como array de caracteres***

Na linha 1 da figura 10, declaramos um array de **char** *input* de 4 posições. Na linha 7 da mesma figura, lemos da entrada padrão valores que serão atualizados na variável *input*. Essa leitura é feita como string, isto é, leva em consideração que os valores cedidos à entrada padrão serão seqüências de caracteres (ou seja, strings).

2.3. Sistema de Tipos

A linguagem C é estaticamente tipada (toda variável e parâmetro possui um tipo fixo escolhido pelo programador) e sua checagem de tipos é fraca, isto é, é possível realizar livremente coerção entre vários tipos da linguagem (sem ocasionar erro) – o tópico sobre coerção será expandido na seção 10.3. Como exemplo, basta observarmos que a soma de duas letras (caracteres), atribuída a uma variável do tipo **float** não resulta em erro de compilação, como na figura 11.

```
1  char a,b;
2  float c;
3  a = 'a';
4  b = 'b';
5  c = a+b;
```

Figura 11 – ***Exemplo de checagem fraca de tipos***

2.4. Classificação de Valores

Entre os valores existentes em C, somente aqueles de tipos primitivos, variáveis heap e registros são valores de primeira classe. Arrays são considerados valores de segunda classe, uma vez que só é possível passá-los por parâmetro ou retorná-los de função mediante referenciação por variáveis heap ou via definição de seu tamanho. Os valores de string, por ser também um array, são conseqüentemente valores de segunda classe.

Na primeira linha da figura 12, vemos que o array é referenciado pelo seu primeiro elemento, cujo apontador é um dos parâmetros formais da função. O outro parâmetro é o comprimento do array: não é possível sabermos qual seu comprimento total uma vez que conhecemos somente seu primeiro elemento. O array é percorrido como indicado na linha 6, através de aritmética de apontadores: quando se soma uma quantidade X a um ponteiro, o endereço armazenado é incrementado em X vezes o comprimento do tipo apontado por essa variável heap. Mais sobre variáveis heap pode ser visto na seção 4.4.

```
1  int getSum(int *first_element, int size) {
2     int i = 0;
3     int sum = 0;
4     for(i = 0; i < size; i++) {
5        sum += *first_element;
6        first_element++;
7     }
8     return sum;
9  }
```

Figura 12 – ***Parametrização de array via apontador***

Por possuir valores de segunda classe, concluímos que C viola o Princípio da Completude de Tipos.

3. Expressões

3.1. Expressões Agregadas

A linguagem C admite o uso de expressões agregadas para atribuições de variáveis dos tipos compostos produto cartesiano (registro) e mapeamento (array).

```
1   struct _Jogador {
2         char * nome;
3         Pos (*jogar)();
4         int acertos;
5         TipoCasa casaBomba;
6   };
7
8   struct _Jogador criarHumano() {
9      struct _Jogador humano = { .nome = "Humano",
10                                .jogar = &jogarHumano,
11                                .casaBomba = bombaHumano };
12     return humano;
13  }
```

Figura 13 – ***Expressão agregada sobre produto cartesiano***

Nas linhas 1-6 da figura 13, vemos a declaração do tipo **Jogador**, um registro de quatro campos. Nas linhas 9-11, podemos ver o registro *humano*, do tipo **Jogador**, sendo inicializado por uma expressão agregada. Vemos que cada campo é identificado (*nome*, *jogar*, *casaBomba*) de modo a podermos especificar qual o seu valor de inicialização. Isso ainda permite que haja independência na ordem de inicialização dos campos, bem como não obriga que todos os campos do produto cartesiano sejam inicializados na expressão agregada.

```
1   struct constante {
2       int arredondado;
3       float valor;
4       char * nome;
5   };
6
7   struct constante pi = { 3, 3.1415, "Pi" };
8
9   struct constante e = {
10      .nome = "Numero de Euler",
11      .valor = 2.7182,
12      .arredondado = 3
13  };
14
15  int array[6] = { 1, 2, 3, 4, 5, 6 }
```

Figura 14 – ***Expressões agregadas em C***

Podemos conferir a independência de ordem de inicialização dos campos na expressão agregada presente nas linhas 9-13 da figura 14: o produto cartesiano, declarado nas linhas 1-5, foi inicializado sem levar em consideração a ordem de declaração de seus campos. Na linha 7 ainda da figura 14, no entanto, o produto cartesiano está sendo inicializado mediante uma expressão agregada que não define os campos inicializados. Isso atrela os valores da expressão à ordem de declaração dos

campos no produto cartesiano. Ainda, se for necessário deixar algum campo sem inicialização, todos os campos subseqüentes não poderão ser inicializados.

Na linha 15 da figura 14, vemos uma expressão agregada para inicialização de um mapeamento array (como comentado na seção 2.2.3). Um array de inteiros é definido através de uma expressão agregada, cujo valor inicializará a variável *array*. A ordem dos valores é levada em consideração: o primeiro valor é referente à posição zero, e cada valor que se segue refere-se à posição do valor à esquerda acrescido de um. O total de valores definido na expressão agregada não pode exceder o tamanho do array.

3.2. Expressões Condicionais

Uma única forma de expressão condicional é permitida em C. Ela é chamada popularmente de *if-ternário*, e é composta por 3 expressões: uma expressão de teste e duas expressões de retorno, na seguinte forma:

```
expTeste ? expRetornadaVerdade : expRetornadaFalso;
```

A expressão de teste (situada à esquerda do sinal de interrogação) é avaliada e, caso seu valor booleano seja verdadeiro, o valor da avaliação da expressão de veracidade (situada entre o sinal de interrogação e os dois-pontos) é assumido como o valor de avaliação de toda a expressão condicional; caso o valor booleano seja falso, o valor da avaliação da expressão de falsidade (situada á direita dos dois-pontos) é assumido como o valor de avaliação de toda a expressão condicional.

As expressões condicionais em C possuem uma grande vantagem: sua simplicidade. O corpo da expressão é bem compacto, sem pecar em legibilidade. Vejamos como ficaria a expressão condicional anterior se fosse escrita em pseudocódigo:

```
IF teste THEN
   expRetornadaVerdade
ELSE
   expRetornadaFalso
```

```
1  Jogador * atual = &humano;
2
3  (...)
4
5  atual = (atual == &humano) ? &computador : &humano;
```

Figura 15 – Uso de e*xpressão condicional em C*

Na linha 5 da figura 15, uma expressão condicional é avaliada para atualizar a variável *atual*, declarada na linha 1 da mesma figura. De acordo com a expressão de teste, caso o valor armazenado em *atual* seja *&humano*, seu valor deverá ser atualizado para *&computador*; caso seja *&computador*, para *&humano*. Atentemos ao fato de que os valores das avaliações tanto da expressão de veracidade quanto da de falsidade possuem mesmo tipo que o da variável *atual*, de modo que não há problemas de compatibilidade de tipos nenhum dos casos de atualização (expressão de teste ser verdadeira ou falsa).

4. Comandos

4.1. Comandos Condicionais

Em C, os comandos condicionais devem ser parametrizados por expressões booleanas ou inteiras. São suportados dois tipos de comandos condicionais: o **if** (expressão booleana) e o **switch** (expressão inteira).

```
1  for (i = 0; i < campo.tamanho; i++) {
2     if (i < 9) {
3        printf("%d  ", i); // dois espaços à direita do int
4     } else {
5        printf("%d ", i); // um único espaço à direita do int
6     }
7  }
```

Figura 16 – *Comando condicional if*

O comando condicional ***if*** pode ser visto na figura 16, nas linhas 2 a 6. A expressão avaliada ($i < 9$) possui valor booleano. Caso seja verdadeira, a seqüência de comandos delimitada pelo bloco do ***if*** será executada; caso seja falsa, a seqüência delimitada pelo bloco do ***else*** o será.

```
1  enum TipoCasa {zero, uma, duas, tres, quatro, cinco, seis};
2
3  enum TipoCasa tipo_casa = zero;
4  char c;
5
6  switch(tipo_casa) {
7     case zero: {
8        c = ' ' ;
9        break;
10    }
11    case uma: {
12       c = '1';
13       break;
14    }
15    case duas: {
16       c = '2';
17       break;
18    }
19    default: {
20       c = '#';
21    }
22 }
```

Figura 17 – *Comando condicional switch*

O comando condicional **switch** pode ser visto na figura 17, nas linhas 6 a 22. Na linha 6 é especificada a variável que será avaliada. Cada um dos casos, definidos nos blocos 7-10, 11-14, 15-18 e 19-21, determina uma seqüência de comandos a serem executados caso o valor avaliado da variável seja igual ao definido pelo caso. Quando o valor satisfizer o caso, todos os comandos do caso satisfeito serão executados, além de todos os comandos dos casos subseqüentes. Comandos definidos dentro do bloco do

switch mas fora de qualquer caso não serão executados. O comando condicional **switch** termina ao final do último caso, ou caso seja executado o seqüenciador **break**: o fluxo de execução salta para o primeiro comando após o final do bloco do **switch**.

O caso especial **default**, presente nas linhas 19-21 da figura 17, especifica o caso em que o valor não foi condizente com nenhum dos casos já testados. Isso significa que um valor que não satisfizer nenhum dos casos anteriores sempre vai satisfazer o caso **default**. Pode haver no máximo um caso **default** em cada **switch**.

A linguagem C não suporta comandos condicionais não-determinísticos.

5. Variáveis

5.1. Variáveis Armazenáveis

Todos os valores definidos pelos tipos primitivos existentes na linguagem são armazenáveis. Além disso, podemos armazenar endereços de memória para outras variáveis, abstrações de funções e de procedimentos. Valores de tipos compostos não são armazenáveis por poderem ser atualizados seletivamente.

5.2. Atualização

Tanto a atualização total quanto a atualização seletiva de uma variável composta são permitidas em C. O tipo dessa variável pode tanto ser um produto cartesiano quanto um mapeamento array.

```
1  struct _Pos {
2    int m, n;
3  };
4
5  struct _Pos posAtual = { 0, 0 };
6
7  struct _Pos novaPos = {.m = -1, .n = -1};
8
9  novaPos = posAtual;
10 novaPos.m = posAtual.m + 1;
```

Figura 18 – ***Atualizações total e seletiva sobre registros***

Na figura 18, podemos observar atualizações de variáveis compostas de tipo produto cartesiano. Na linha 5, atentamos à declaração da variável *posAtual*, acompanhada de uma atualização total através de uma expressão agregada. Na linha 7, há a declaração da variável *novaPos*, também atualizada totalmente mediante uma expressão agregada, cuja sintaxe difere do agregado definido na linha 5. Na linha 9 da figura 18, vemos um exemplo de atualização total da variável composta *novaPos*. Todo o valor da variável *posAtual* é copiado para a variável *novaPos*, ou seja, os componentes das duas células de armazenamento de *posAtual* são copiados para as duas células de armazenamento de *novaPos*. Ainda na mesma figura 18, na linha 10, podemos ver que um único campo da variável *novaPos* é atualizado, deixando o outro componente intacto. Este é um exemplo de atualização seletiva na linguagem C.

```
1  int array[3];
2  int outro_array[3];
```

```
3
4  outro_array = {0, 1, 2};
5  array = outro_array;
6  array[1] = outro_array[2];
```

Figura 19 – ***Atualizações total e seletiva sobre arrays***

Na figura 19, observamos atualizações de variáveis que mapeiam **int** em **int**. Na linha 4, atualizamos totalmente a variável *outro_array* (declarada na linha 2) por meio de uma expressão agregada. Na linha 5, realizamos outra atualização total: a variável *array* é atualizada com os valores definidos na variável *outro_array*. Na linha 6 da figura 19, ocorre uma atualização seletiva do componente da posição 1 da variável *array*, que receberá o valor avaliado do componente de posição 2 da variável *outro_array*.

5.3. Arrays

A linguagem C implementa nativamente arrays estáticos. A figura 20 mostra a declaração e atribuição de valores a um array estático: um array estático é declarado na linha 1, e os valores de cada um dos seus componentes são atualizados nas linhas seguintes.

```
1  int array[3];
2  array[0] = 5;
3  array[1] = 2;
4  array[2] = 7;
```

Figura 20 – ***Instanciação de array estático***

É possível simular arrays dinâmicos na linguagem C através do uso de apontadores e das operações de alocação e desalocação de memória.

```
1  Campo campo;
2  campo.tamanho = n;
3  campo.tabuleiro = malloc(sizeof(Casa) * (n + 2));
   free(campo.tabuleiro);
```

Figura 21 – Simulação de ***array dinâmico***

O tamanho do array requisitado é definido no momento de sua alocação. Na linha 3 da figura 21, foram pedidas n + 2 posições de componentes **Casa**, que são agora referenciadas pela variável *campo.tabuleiro*. Por fim, na linha 4 da mesma figura, essas posições são liberadas de volta para a memória. Para maiores detalhes sobre ponteiros, ver seção 5.4.

5.4. Variáveis Heap (Apontadores)

Variáveis heap são implementadas nativamente pela linguagem C. Ponteiros ou apontadores, como também são conhecidas as variáveis heap, armazenam na memória endereços para outras posições da memória – seja para variáveis de quaisquer tipos existentes, seja para abstrações de função ou procedimento.

Para lidar com ponteiros, além dos operadores aritméticos e de acesso a componentes referenciados (via índice ou seta), faz-se uso de dois operadores unários: o asterisco "*" e o *ampersand* "&" (e-comercial). O asterisco possui dois tipos de uso: sucede um tipo e precede um identificador em declarações de ponteiros para um dado

tipo; e precede um identificador de ponteiro, resultando em uma expressão que, ao ser avaliada, resulta no valor armazenado pela variável referenciada pelo ponteiro. O *ampersand*, por outro lado, pode ser usado somente precedendo variáveis; essa expressão, quando avaliada, resulta em um valor inteiro que representa o endereço de memória alocado para armazenar os valores daquela variável..

A figura 22 mostra como é feita a declaração e atribuição de uma variável heap em C a uma variável já existente.

```
1  int * inteiro;
2  int outro_inteiro = 2;
3  inteiro = &outro_inteiro;
```

Figura 22 – ***Declaração e atribuição de apontador para inteiro***

Na linha 1 da figura 22, é declarada uma variável heap de identificador *inteiro* para o tipo int com uso de asterisco. Na mesma figura, na linha 3, atribuímos ao apontador *inteiro* o endereço da variável *outro_inteiro*, declarada e atribuída na linha 2.

Para se utilizar o espaço referenciado pelo apontador, este espaço deve estar previamente alocado. Caso não já seja referenciado por outra variável, faz-se necessária a alocação de um espaço na memória. Essa operação é feita através da abstração de função **malloc** (operação alocadora). Essa abstração de função recebe por parâmetro o tamanho (em bytes) do espaço de memória a ser alocado, e a avaliação da expressão resulta no endereço de memória alocado com o tamanho especificado.

```
1  int i;
2  for (i = 0; i <= n + 1; i++) {
3     campo.tabuleiro[i] = malloc(sizeof(Casa) * (n + 2));
4     if (campo.tabuleiro[i] == NULL) {
5        (...)
6     }
7  }
```

Figura 23 – ***Alocação de espaço em memória para apontadores***

Na figura 23, observamos alocação de espaço na memória para cada uma das variáveis heap pertencentes ao array *campo.tabuleiro*. A atribuição do espaço alocado é feita na linha 3 da mesma figura. O tamanho do espaço requisitado foi definido com auxílio da abstração de função **sizeof**: ela recebe um tipo e resulta no tamanho em bytes necessário para armazenar os valores daquele tipo. Dessa maneira, podemos alocar espaço necessário para armazenar quaisquer valores, ou até mesmo uma lista de valores: se multiplicarmos o espaço necessário para armazenar um único valor, podemos armazenar diversos valores (simulação de array dinâmico – ver seção 5.3).

A expressão **malloc** reserva na memória um espaço contíguo para armazenar os valores de tamanho definido. Caso não seja possível alocar tal espaço, a função resulta em NULL (espaço reservado na memória que indica a inexistência de referência). É comum realizar testes de nulidade de referência antes de começar a utilizar o espaço alocado, para evitar erros em tempo de execução (uma vez que o espaço NULL é livremente usado e referenciado).

Uma vez que o uso da variável heap foi finalizado, é recomendada a desalocação do espaço referenciado por ela. Esse espaço não é desalocado automaticamente, como é feito para outros tipos de variáveis; para isso, existe o comando **free**. Esse comando

recebe por parâmetro a variável heap cujo espaço referenciado será desalocado. Ao final de sua execução, o espaço referenciado foi desalocado.

```
1  #include <stdlib.h>
2
3  void desalocarCampo(Campo campo) {
4     int i;
5     for (i = 0; i <= campo.tamanho + 2; i++) {
6        free(campo.tabuleiro[i]);
7     }
8     free(campo.tabuleiro);
9  }
```

Figura 24 – *Desalocação de espaço em memória para apontadores*

Na figura 24, observamos desalocação de espaço em memória referenciado por variáveis heap. Inicialmente, nas linhas 5 a 7, cada uma das variáveis heap referenciadas por *campo.tabuleiro* é desalocada. Por fim, na linha 8, a própria variável heap *campo.tabuleiro* é desalocada.

Após o espaço referenciado por um ponteiro ser desalocado, a variável é considerada uma "referência solta", uma vez que aponta para um espaço na memória que não está dedicado ao ponteiro e que pode estar sendo referenciado por um outro processo em execução. Para evitar sobrescrever valores de outros processos, as referências soltas costumam ser atualizadas para o espaço NULL: qualquer mudança no valor do espaço referenciado por essa variável não acarretará em dano aos demais processos em execução no sistema.

Para utilizar as operações **malloc** e **free**, é necessário indicar a localização da implementação dessas operações. Isso é feito através da diretiva de pré-compilação **#include**, e indicando a biblioteca padrão **stdlib.h** – como feito na linha 1 da figura 24.

5.5. Variáveis Persistentes

A linguagem C oferece o tipo **FILE** para lidar diretamente com dados persistentes. Na figura abaixo, *FILE* representa uma abstração do arquivo. A função *fopen(**char** * nome do arquivo, **char** * modo)* retorna *null* caso o arquivo não possa ser aberto ou não tenha sido encontrado.

No Campo Minado não foram usadas variáveis persistentes.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main () {
5     FILE* f;
6     f = fopen("arquivo.txt", "rt");
7
8     if (f == NULL) {
9        printf("Erro na abertura do arquivo.");
10       getchar();
11       exit(1);
12    }
13
14    printf("\n Se chegou nesta linha o arquivo foi aberto!");
15    getchar();
```

```
16      fclose(f);
17  }
```

Figura 25 – ***Variáveis persistentes em C***

6. Associações

Associações podem ser declaradas em qualquer parte de um programa C, e seus identificadores só poderão ser usados depois de realizada a associação. Um mesmo identificador pode representar diferentes entidades, independentemente do total de vezes que for declarada ou de sua localização no programa. Em C, um identificador pode ser declarado somente uma única vez em cada ambiente.

```
1   int smth = 10;
2
3   double see();
4
5   int main() {
6       char smth;
7       return 0;
8   }
9
10  double see() {
11      double smth = 2;
12      return smth;
13  }
```

Figura 26 – ***Diferentes ambientes em um programa C***

Declarações em C seguem como descrito nas linhas 1, 6 e 11 da Figura 26. A interpretação das entidades representadas pelo identificador *smth* nessa figura depende do ambiente em que ele está localizado.

Um tipo é associado ao identificador, e pode ser acompanhado de uma atribuição de valor à entidade que esse representa.

6.1. Entidades Associáveis

Em C, valores primitivos, compostos e tipos, até expressões condicionais e declarações de variáveis são associáveis. Essas associações podem ser definidas como visto na figura 22.

```
1   #include <stdio.h>
2
3   #define cond (1 > 2)
4   #define showOnDefOut(a) printf(a)
5
6   int getTen();
7
8   #define charType char
9
10  int main() {
```

```
11      int (*ptrToGet10)() = getTen;
12      charType* string = "string";
13
14      if(!cond) {
15          showOnDefOut("Condition is true.\n");
16      }
17
18      return 0;
19  }
20
21  int getTen() {
22      int zero = 0;
23      #define ten 10
24      return zero + ten + zero;
25  }
26  const double PI = 3.14159;
```

Figura 27 – ***Associações entre entidades e identificadores***

As associações mais simples são aquelas feitas por definição de macros, através de **#define**. Nas linhas 3, 4, 8 e 23, da figura 27, vemos associações de identificadores a expressão condicional, abstração de função, tipo primitivo e valor constante, respectivamente. Na linha 6 vemos uma associação entre um identificador e uma abstração de função. Nas linhas 11, 12 e 22, temos associações entre identificadores e identificador de abstração de função, valor composto e valor primitivo, respectivamente. Na última linha, também temos uma associação entre um identificador e um valor primitivo.

6.2. Blocos

Blocos em C são geralmente delimitados por chaves. Uma chave de abertura ("{") determina o início do bloco, enquanto uma chave de término ("}"), o fim do bloco. Comandos-blocos e expressões-blocos podem não necessitar de delimitadores caso possuam somente uma chamada em seu bloco.

Na figura 27, as linhas 14-16 determinam o comando-bloco **if**, e os delimitadores de início e de fim do bloco (cujo único subcomando é **showOnDefOut**) são os últimos caracteres das linhas 14 e 16, respectivamente.

```
1   #include <stdio.h>
2
3   int main() {
4      char a = 'a';
5      printf("%c\n", a);
6      return 0;
7   }
```

Figura 28 – ***Bloco principal "main"***

Na figura 28, vemos o código de um programa executável. Todo o código executável deve estar situado em um bloco definido pela função **main** (linhas 3 a 7). Um programa executa seqüencialmente as operações definidas dentro do bloco **main** ao ser rodado.

6.3. Estrutura de blocos

Blocos podem ocorrer dentro de blocos, como já visto. Essa estrutura de blocos aninhada permite que um identificador esteja associado a diferentes entidades em cada um dos ambientes.

```
1  #include <stdio.h>
2
3  int globalVar = 1;
4
5  procedure1() {
6     int proc1Var = 2;
7     printf("Running procedure 1...\n");
8  }
9
10 int main() {
11    int mainVar = 3;
12    printf("Running main...\n");
13
14    void procedure2() {
15       int proc2Var = 4;
16       printf("Running procedure 2...\n");
17    }
18
19    procedure1();
20    procedure2();
21
22    if(1) {
23       int ifVar = 5;
24       printf("Running if block...\n");
25
26       void procedure3() {
27          int proc3Var = 6;
28          printf("Running procedure 3...\n");
29       }
30       procedure3();
31
32       {
33           int blockVar = 7;
34           printf("Running a generic block...\n");
35       }
36    }
37
38    return 0;
39 }
```

Figura 29 – ***Aninhamento de blocos***

A linguagem C não restringe a ocorrência de blocos aninhados. Desse modo, podemos sempre definir um bloco interno a outro bloco, como podemos ver na figura 29. Temos dois blocos mais externos -- um procedimento (linhas 5 a 8), e a função principal do programa (linhas 10 a 39) -- e, interior à função principal do programa, temos dois outros comandos-blocos -- linhas 14 a 17 e 22 a 36. Interior ao comando-bloco **if**, temos outros dois blocos interiores: um procedimento, que se estende da linha

26 à linha 29; e um bloco genérico (que não é relativo a comando ou expressão), da linha 32 à 35. Cada um desses blocos delimita o escopo das variáveis declaradas dentro dele.

Recomenda-se o uso de um identificador para cada entidade associada, a fim de evitar qualquer tipo de ambigüidade que possa ocorrer.

6.4. Escopo

C suporta escopo estático. Desta forma, o corpo de um bloco é avaliado em tempo de compilação, como podemos ver no código abaixo:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int x = 10; //Global
5  int y = 15; //Global
6  void f() {
7       if (y > x) {
8          int z = x + y;
9          printf("F: z = x+y = %d\n",z);
10      }
11 }
12 void g() {
13      int w;
14      w = x;
15      printf("G: w = x = %d\n",w);
16 }
17 int main() {
18      f();
19      int x = 14;
20      g();
21      system("pause");
22      return 0;
23 }
```

Figura 30 – ***Escopo***

Como é visto na figura 30, há a inicialização da variável local *x*, cujo nome é o mesmo da variável global. Com a chamada de *g*, é impresso o valor 10 referente à variável global *x*, o que caracteriza escopo estático.

6.5. Definições

Em C, a definição de constantes é realizada através do **#define**.

```
#define nomeDaConstante valorDaConstante;
```

Como no Campo Minado, o tamanho máximo do tabuleiro é uma constante.

```
1  #define TAM_MAX 23
```

Figura 31 – ***Definição de constante***

6.5.1. Declaração e definição de tipos

A declaração de tipos pode ser feita com **structs**, onde a cada ocorrência de uma **struct** é criado um novo e distinto tipo. No exemplo da figura 32 o nome do novo tipo é definido em "*nome_do_tipo*", e os tipos dos possíveis elementos associados ao novo tipo são definidos onde, no código, há "*tipo_do_elemento*" e "*tipo_dos_elementos*".

```
1  struct nome_do_tipo
2  {
3     tipo_do_elemento a;
4     tipo_dos_elementos b, c;
5
6     ...
7  };
```

Figura 32 – *Declaração de um novo tipo*

Para a definição de tipos, pode ser usada a palavra reservada **`typedef`**, que associa um novo identificador a um tipo já existente, como mostrado em 2.5.

```
1  #include <stdlib.h>
2  #include <stdlib.h>
3
4  typedef int inteirao;
5  int main()
6  {
7     inteirao quatro = 4;
8     int cinco = 5;
9     inteirao nove = quatro + cinco;
10    printf("Soma de quatro e cinco: %d", nove);
11    system("pause");
12    return 0;
13 }
```

Figura 33 – *Definição de tipo*

Na figura 33, a palavra reservada **`typedef`** seguida do tipo já existente e logo após o novo identificador (linha 4) aplica, em C, o conceito de definição de tipos.

6.5.2. Declaração de variáveis

A declaração de variáveis pode ser feita com imediata atribuição de valor ou não. Para declarar uma variável sem defini-la, ou seja, sem fazer com que o compilador reserve um espaço na memória para o seu conteúdo, é necessário o uso da palavra reservada **extern**, como se pode ver abaixo:

```
1  (...)
2  int a;
3  int b = 3;
4  double d = (1024*1024)/16;
5  extern long long h;
6  extern Jogador humano;
7  (...)
```

Figura 34 – *Declaração de variáveis*

Nas linhas 2, 3 e 4, na figura 34, as variáveis são declaradas e imediatamente definidas. Nas linhas seguintes, com o uso da palavra reservada "**extern**", houve apenas a declaração da variável *h* e *humano*, sem alocação de espaço na memória.

Tomando como base a figura 32, podemos também declarar variáveis para o tipo genérico mostrado nesta, como mostrado na figura 35.

```
1  ...
2  struct nome_do_tipo variavel;
3  ...
```

Figura 35 – *Declaração de variável utilizando tipo definido por usuário*

Na figura 35, vemos um exemplo de declaração da variável *variavel*, associando-a ao tipo composto *nome_do_tipo*.

6.5.3. Declarações seqüenciais

C suporta declarações seqüenciais.

```
1  int totBombas;
2  totBombas = totBombas + 1;
```

Figura 36 – *Declaração seqüencial*

Declarações seqüenciais são feitas como definido na figura 36. Utiliza-se o caractere ponto-e-vírgula ";" para especificar seqüencialidade das declarações feitas em C. Esse caractere é pode ser verificado ao final das linhas 1 e 2 da figura 36.

6.5.4. Declarações colaterais

C não suporta declarações colaterais.

6.5.5. Declarações recursivas

C suporta declarações recursivas, com a única restrição de que o identificador tenha sido declarado antes de ser usado.

```
1  void abrirCasa(Campo campo, Pos pos, Jogador *jogador) {
2          Casa * casa = getCasa(campo, pos);
3          posAdjacente.n = pos.n + 1;
4          abrirCasa(campo, posAdjacente, jogador);
5  }
```

Figura 37 – *Declaração recursiva*

Na figura 37, o identificador usado recursivamente é *abrirCasa*. Ele é declarado na linha 1 da mesma figura, e invocado novamente na linha 4, ainda dentro do bloco que especifica a implementação da abstração de procedimento.

6.5.6. Comandos-Bloco

C dá suporte ao uso de comandos-bloco.

```
1  if (casa->tipoCasa == zero) {
2     Pos posAdjacente;
3     posAdjacente.m = pos.m - 1;
4     (...)
5     posAdjacente.n = pos.n + 1;
6     abrirCasa(campo, posAdjacente, jogador);
7  }
```

Figura 38 – ***Comando-bloco***

Na figura 38, vemos um comando-bloco condicional **if**. O comando se estende da linha 1 à linha 7. Para consistir um comando-bloco, é necessário declarar uma variável interior ao bloco definido pelo comando, que auxiliará somente na execução do comando. Podemos ver essa declaração na linha 2 da figura 38: uma variável do tipo **Pos** e identificador *posAdjacente*.

6.5.7. Expressões Bloco

C suporta expressões-bloco.

```
1  Pos chuteRandomico(Campo campo) {
2     Pos pos;
3     do {
4        pos.m = (int) (((double) rand() / RAND_MAX) *
5                       (campo.tamanho - 1));
6        pos.n = (int) (((double) rand() / RAND_MAX) *
7                       (campo.tamanho - 1));
8     } while (!podeTerBomba(campo, pos));
9     return pos;
10 }
```

Figura 39 – ***Expressão-bloco***

Uma abstração de função define uma expressão-bloco, como a abstração *chuteRandomico* da figura 39. Os blocos estão delimitados pelos últimos caracteres das linhas 1 e 10. Para se consistir uma expressão-bloco, vemos uma declaração da variável de identificador *pos* na linha 2 da mesma figura. Essa variável vai ser utilizada somente para avaliar a expressão *chuteRandomico*, e desalocada logo finda a avaliação.

7. Abstrações

Em C não existe distinção entre abstrações de função e de procedimento. Cabe ao programador definir, na implementação, qualquer abstração de procedimento como sendo uma abstração de função que não retorna valor algum.

Estruturalmente, uma abstração de função é definida de acordo com a figura 40 e uma abstração de procedimento é definida similarmente ao exemplo mostrado na figura 41.

```
1  tipo_de_retorno identificador_da_funcao (tipo1 param1, tipo2
2  param2, ... , tipoN paramN) {
3           //corpo da função
4           return valor_de_retorno;
5  }
6
```

Figura 40 – ***Estrutura de uma abstração de função***

Na figura 40, a função, cujo identificador é descrito onde se encontra *identificador_da_funcao*, retorna um valor do tipo definido em *tipo_de_retorno*. Entre parênteses estão os parâmetros da função, precedidos cada um pelo seu tipo, e dentro do seu bloco está a sua implementação seguida do valor que é retornado (a palavra reservada **return** precede o valor de retorno).

É importante salientar que para implementarmos uma abstração de procedimento, devemos atribuir **void** ao tipo de retorno, para declararmos que a função não retorna nada.

```
1  #include <stdio.h>
2  void swap(int*,int*, char);
3
4  main()
5  {
6       int a=20, b=30;
7       swap(&a,&b);
8       printf("a=%d e b=%d",a,b);
9       return 0;
10 }
11
12 void swap(int *x, int *y, char c)
13 {
14      printf("c = %d",c);
15      int aux;
16      aux = *x; *x = *y; *y = aux;
17      return;
18 }
```

Figura 41 – ***Exemplo de abstração de procedimento***

A abstração de procedimento *swap*, na figura 41, tem como parâmetros, dois apontadores que apontam para valores do tipo **int** na memória. Como a passagem de parâmetros é por cópia, os apontadores continuam referenciando os mesmos valores, o que possibilita mudar o conteúdo destes.

Valores compostos e abstrações não podem ser argumentos diretos em C. Já os apontadores e valores primitivos podem, como vemos na figura 36. Entretanto, valores compostos e abstrações podem ser passados por parâmetro através de apontadores para estes elementos.

8. Passagem de parâmetros

Apesar da linguagem C não dar suporte a mecanismos de passagem de parâmetros diretamente, C não viola o princípio de correspondência, pois se pode atingir o mesmo efeito utilizando ponteiros.

8.1. Passagem de parâmetros por cópia

C suporta a passagem de parâmetros por cópia, sendo possível apenas a passagem por valor.

```
1  int dobro(int x){
2     x = x*2;
3     return x;
4  }
5
6  int main(){
7     int v1;
8     int v2 = 3;
9     v1 = dobro(v2);
10 }
```

Figura 42 – ***Passagem de parâmetro por cópia***

Na figura 42, quando a linha 9 é executada o valor de *v2* não se altera.

8.2. Passagem de parâmetros por definição

C não suporta passagem de parâmetro por definição. É possível, no entanto, simular passagem de parâmetro por definição através de apontadores.

9. Encapsulamento

Os únicos tipos de módulos que a linguagem C suporta são abstrações, que foram discutidas na sessão 7, e pacotes.

9.1.1. Pacotes

Pacotes em C são suportados através dos arquivos de cabeçalho (*header files*). Um *header file* é um arquivo, geralmente no formato **.h**, que contém declarações de constantes, variáveis e abstrações, além de definições de tipos. Esse arquivo de cabeçalho é associado a um arquivo de código contendo as definições das variáveis e das abstrações que foram declaradas. Desta forma, um arquivo de cabeçalho funciona como uma interface do arquivo de código ao qual ele se refere, permitindo o uso apenas dos componentes que foram exportados através dele.

Os componentes de um arquivo de cabeçalho podem ser acessados num programa através da inclusão do pacote, com o uso da diretiva de pré-processamento **#include**. Para clarificar o entendimento, observe os exemplos a seguir:

```
1  /* Arquivo de cabeçalho matematica.h */
2
3  typedef int point[2];
4
```

```
5  const double pi = 3.1415;
6
7  /* palavra-chave extern indica variável global */
8  extern int count;
9
10 int somaInteiros(int, int);
```

Figura 43 – ***Arquivo de cabeçalho matematica.h***

A figura 43 mostra a estrutura do arquivo de cabeçalho *matematica.h*. Este pacote define os componentes que são acessíveis para quem o inclui.

```
1  /* Arquivo matematica.c */
2
3  #include "matematica.h"
4  int count = 3;
5  int aux = 2;
6
7  int somaInteiros(int a, int b) {
8     return a + b;
9  }
10 int somaDoubles(double c, double d) {
11    return c + d;
12 }
```

Figura 44 – ***Arquivo de código matematica.c***

As definições da variável e da função declarados em *matematica.h* são encontradas no arquivo *matematica.c*, mostrado na figura 44. Observe que este arquivo também deve incluir o arquivo *matematica.h*.

```
1  /* Arquivo teste.c */
2  #include "matematica.h"
3
4  int triplo(int x) {
5     return somaInteiros(x, soma(x, x));
6  }
7
8  // retorna true - variável declarada no cabeçalho importado!
9  boolean teste(){
10    return count == 3;
11 }
```

Figura 45 – ***Arquivo que inclui pacote matematica.h***

A figura 45 mostra o arquivo *teste.c*, que inclui o arquivo de cabeçalho *matematica.h*, ficando então apto a utilizar todos os elementos deste pacote. No momento da compilação de *teste.c*, deve ser indicado o arquivo aonde se encontram as definições do cabeçalho, que neste caso é o arquivo *matematica.c*.

```
1  /* Arquivo erro.c */
2
3  #include "matematica.h"
4
5  /* erro de compilação - função não definida! */
```

```
6  double triplo(double y){
7     return somaDoubles(y, soma(y,y));
8  }
9
10 /* erro de compilação - variável não definida! */
11 boolean teste(){
12    return aux == 3;
13 }
```

Figura 46 – ***Arquivo que tenta acessar elementos inacessíveis do pacote***

Por fim, o arquivo *erro.c*, mostrado na figura 46, tenta utilizar componentes do arquivo *matematica.c* que não foram exportados pelo arquivo *matematica.h*, provocando erros de compilação.

Os arquivos de cabeçalhos são bastante úteis na criação e no uso de bibliotecas.

9.1.2. TADs

C não dá suporte a TADs.

9.1.3. Objetos

C não suporta objetos e classes.

10. Sistema de tipos

A linguagem C não é puramente monomórfica. Apesar de não existir nenhuma abstração de usuário que dê suporte a polimorfismo paramétrico, muitas abstrações *built-in* são sobrecarregadas ou polimórficas.

10.1. Sobrecarga

A linguagem C dá suporte a sobrecarga, porém somente operações *built-in* podem ser sobrecarregadas. Operações conflitam se forem declaradas mais de uma vez, no mesmo ambiente, com o mesmo identificador. Operações sobrecarregadas em C são independentes de contexto.

10.2. Polimorfismo

A linguagem C suporta polimorfismo somente em operações *built-in*. É possível, no entanto, simular polimorfismo, seja através de tipos compostos, ou conversão explícita de tipos.

10.3. Coerção

A linguagem C, por ser uma linguagem fracamente tipada, é bastante permissiva no que tange a coerções. Por exemplo, C permite coerções de inteiros para números reais, de inteiros de baixo alcance para inteiros de alto alcance, de números reais de baixa precisão para reais de alta precisão e de caracteres para inteiros ou reais.

```
1  double a = 'a'; /* valor real 97.0 */
2
3  int b = 3.2 + 'a'; /* valor inteiro 100 */
4
5  char c = 196; /* caractere - (ASCII 97), borda superior do
6  tabuleiro */
7
   long d = c; /* valor long 100 */
```

Figura 47 – *Coerção em C*

Através dos exemplos na Figura 47 – Coerção em C é possível perceber a variedade de conversões implícitas de tipo permitidas por esta linguagem.

10.4. Subtipos e herança

C não permite ao usuário definir subtipos e herança. Entretanto, subtipos primitivos da linguagem apresentam herança. Por exemplo, os tipos inteiros **int** e **char** podem ser vistos como subtipos de **long**, e o tipo **float** pode ser visto como subtipo de **double**. Dessa forma, esses tipos herdam todas as operações definidas para os seus supertipos, como por exemplo, soma e subtração.

```
1  int a = 1; /* subtipo primitivo de long */
2
3  int b = 2; /* subtipo primitivo de long */
4
5  int c =  a + b; /* herança da operação soma */
6
7  long d = c; /* livre de erros pois int é um subconjunto de
   long */
```

Figura 48 – *Subtipos primitivos*

11. Seqüenciadores

11.1. Desvio incondicional

Desvio incondicional em C acontece sob a forma do comando **goto**. Observe o exemplo na figura 49.

```
1  int i = 1;
2  int k = 0;
3  int x;
```

```
4
5   _L1: if(i > k) {
6      goto _L2;
7   }
8   x = 1;
9   goto _L3;
10  _L2: x = 0;
11  _L3:
```

Figura 49 – *Desvio incondicional*

A restrição existente para o uso de desvio incondicional é que, apesar de poder saltar de um rótulo que esteja mais à frente ou mais atrás no programa, o rótulo e o **goto** devem estar dentro do mesmo bloco. Desvios incondicionais não foram utilizados na implementação do Campo Minado.

11.2. Escape

C dá suporte a escape fazendo uso dos seqüenciadores **`return`**, **`exit`**, **`break`** e **`continue`**.

return: Retorna o resultado de uma função ao programa que a chamou ao encontrar o primeiro *return*.

```
1   Pos jogarComputador(Campo campo) {
2          double probabilidade;
3
4          Pos posMaiorProbabilidade = casaMaiorProbabilidade-
5   BombasAoRedor(campo, &probabilidade);
6
7          if(probabilidade == 0)
8                 return chuteRandomico(campo);
9
10         return possivelBombaAdjacente(campo, posMaiorProbabi-
11  lidade);
    }
```

Figura 50 – *Escape : return*

exit: Causa a finalização do programa.

```
1   Void exit(function);
```

Figura 51 – *Escape : exit()*

break: É utilizado para interromper a execução dentro de um laço ou do comando condicional **switch** (Como foi visto na figura 14). O programa prossegue sua execução com o próximo comando.

```
1   for (;;) {
2   Casa casa = *getCasa(campo, pos);
3   if (!casa.visivel) {
4        break;
```

```
5   }
```
Figura 52 – *Escape : break*

continue: Quando dentro de um laço, o comando **continue** interrompe a execução apenas da iteração atual, não saindo do laço.

```
1   void montarCampo(Campo campo) {
2           Pos posCasaNaoBomba;
3           for (posCasaNaoBomba.m = 0; posCasaNaoBomba.m < cam-
4   po.tamanho; posCasaNaoBomba.m++) {
5                   for (posCasaNaoBomba.n = 0; posCasaNaoBomba.n
6   < campo.tamanho; posCasaNaoBomba.n++) {
7                           casa = *getCasa(campo,
8   posCasaNaoBomba);
9                           if (ehBomba(casa.tipoCasa)) {
10                                  continue;
11                          }
12                  }
13          }
    }
```
Figura 53 – *Escape : continue*

11.3. Exceções

A linguagem C não trata exceções.

12. Concorrência

A linguagem C permite concorrência entre processos. Desta forma, programas que utilizam concorrência são dependentes de velocidade. Para gerenciar e tentar minimizar essa dependência, a linguagem provê várias bibliotecas que auxiliam na comunicação e sincronização entre processos.

A ocorrência de *deadlocks* e *starvation* não está descartada, por não terem um tratamento especifico (automático) pela linguagem, como por exemplo, o suporte a monitores. Desta forma, a responsabilidade de evitar a ocorrência dessas interações indesejadas fica a cargo do programador. Várias bibliotecas de concorrência em C vão fornecer as primitivas necessárias para que o programador evite a ocorrência *deadlocks* ou *starvation*.

A biblioteca POSIX Threads é amplamente utilizada nos sistemas UNIX e oferece 3 tipos de primitivas para auxiliar na interação entre processos comunicantes. São elas:

- Mutex: Técnica para tratar a sincronização de threads. A terminação "mutex" é uma abreviação de "mutual exclusion", ou exclusão mútua.
- Variáveis de condição: Trata da comunicação de threads que compartilham um mutex.
- Spin-locks: Outra forma de tratar a sincronização de threads, através da espera ocupada.

O uso da biblioteca Pthreads é feito através da inclusão da biblioteca pthread.h. Afora a biblioteca Pthreads, também ilustraremos como a concorrência é gerenciada em C através das bibliotecas semaphore, mpi e rpc.

12.1. Regiões Críticas

A biblioteca Pthreads trata as regiões críticas através da exclusão mútua, que gerencia o acesso exclusivo a essas regiões através das primitivas de interação entre processos: mutex e spin-locks. Além disso, também é possível realizar a exclusão mútua com semáforos, disponibilizados pela biblioteca semaphore.h.

12.1.1. Variáveis mutex

Variáveis mutex foram um dos primeiros meios criados para implementar a sincronização de threads e o acesso a variáveis compartilhadas. Uma variável mutex funciona como uma trava, protegendo o acesso a regiões críticas. O conceito básico de um mutex utilizado em Pthreads é que apenas um processo (thread) pode travar (ou possuir) uma variável mutex em determinado tempo. Dessa forma, se vários processos tentarem travar um mutex para acessar uma região crítica, apenas uma thread vai ter sucesso. Nenhuma outra thread pode ter um mutex até que a thread que o possui libere aquele mutex. Os processos então revezam o acesso aos dados protegidos, realizando assim a exclusão mútua. Uma típica seqüência do uso de mutex segue:

1. Criar e inicializar uma variável mutex
2. Várias threads tentam travar o mutex
3. Apenas uma tem sucesso, e esta possui o mutex
4. A thread que possui o mutex acessa a região crítica livremente
5. A thread que possui o mutex libera o mutex
6. Outra thread adquire o mutex e repete o processo
7. Finalmente o mutex é destruído

Uma variável mutex deve ser do tipo `pthread_mutex_t`, e deve ser inicializada antes de ser utilizada. As principais abstrações para o gerenciamento de mutex são:

Criação e destruição de mutex:

`pthread_mutex_init (mutex,attr)` : Inicializa uma variável mutex. O parâmetro *attr* é opcional;

`pthread_mutex_destroy (mutex)` : Destrói uma variável mutex.

Travar e liberar mutex:

`pthread_mutex_lock (mutex)` : Trava um mutex - bloqueia caso ele já esteja travado;

`pthread_mutex_trylock (mutex)` : Trava um mutex se ele estiver destravado;

`pthread_mutex_unlock (mutex)` : Libera um mutex.

Um exemplo de programa que utiliza mutex é mostrado na figura 54. Duas threads que compartilham uma mesma variável (bignum) são criadas, e o acesso a esta variável (região crítica) é gerenciado através do mutex bignum_mutex. Quando uma thread tenta travar o mutex que já está travado, ela é bloqueada.

```
1  #include <pthread.h>
2
3  void * simple(void *);
4
5  #define NUM_THREADS 2
6  pthread_t tid[NUM_THREADS];      /* array de IDs de
7  threads*/
8
9  int bignum = 0;
10 pthread_mutex_t bignum_mutex;
11
12 main( int argc, char *argv[] )
13 {
14   int i, ret;
15
16   /* Inicializa o mutex */
17   pthread_mutex_init(&bignum_mutex, NULL);
18
19   /* Cria 2 threads simples */
20   for (i=0; i<NUM_THREADS; i++) {
21     pthread_create(&tid[i], NULL, simple, NULL);
22   }
23
24   /* Espera as threads terminarem */
25   for ( i = 0; i < NUM_THREADS; i++)
26     pthread_join(tid[i], NULL);
27
28   /* Destrói o mutex */
29   pthread_mutex_destroy(&bignum_mutex);
30 }
31
32
33 /* Thread simples */
34 void * simple(void * parm)
35 {
36   int i;
37   for(i=0;i<10000;i++) {
38      pthread_mutex_lock (&bignum_mutex) /* acquire -
39 bloqueia se não conseguir*/
40        bignum++;  /* Região crítica */
41      pthread_mutex_unlock(&bignum_mutex); /* relinquish */
42   }
   }
```

Figura 54 – ***Utilização de mutex***

12.1.2. Spinlocks

A biblioteca Pthread especifica operações de spin-lock, mas indica que a sua implementação não é obrigatória. Por este motivo, o suporte nativo a spin-locks vai depender da implementação da biblioteca Pthread que está sendo utilizada.

Spin-locks também são utilizados para realizar a exclusão mútua em regiões críticas, assim como os mutex. Quando se tenta travar um mutex que já está em uso, a thread que tentou adquirir o mutex é bloqueada (pára de executar), enquanto no travamento de um spin-lock em uso, a thread que tentou adquirir o spin-lock fica num loop de espera-ocupada, até o spin-lock ser liberado. As duas soluções têm suas vantagens e desvantagens. Enquanto a espera-ocupada do spin-lock utiliza o processador inutilmente, o custo de bloquear e depois acordar um processo, no mutex, também é considerável. Neste sentido os spin-locks são mais indicados para regiões críticas rapidamente acessadas, enquanto os mutex são mais indicados para regiões críticas que demandam maior tempo de acesso.

Uma variável spin-lock deve ser do tipo `pthread_spinlock_t`, e deve ser inicializada antes de ser utilizada. As principais abstrações para o gerenciamento de spin-locks são:

Criação e destruição de spin-locks:

`pthread_spin_init(lock, pshared)` : Inicializa uma variável spinlock. O parâmetro *pshared* indica se o spin-lock vai ser compartilhado entre processos;

`pthread_spin_destroy(lock)` : Destrói uma variável spinlock.

Travar e liberar spin-locks:

`pthread_spin_lock(lock)` : Trava um spin-lock - entra num loop de espera-ocupada caso ele já esteja travado;

`pthread_spin_trylock(lock)` : Trava um spinlock se ele estiver destravado;

`pthread_spin_unlock(lock)` : Libera um spinlock.

O mesmo exemplo da figura 54 é mostrado na figura 55, agora utilizando spin-locks. Duas threads que compartilham uma mesma variável (bignum) são criadas, e o acesso a esta variável (região crítica) é gerenciado através do spin-lock bignum_spinlock. Quando uma thread tenta travar o spin-lock que já está travado, ela entra num loop de espera-ocupada.

```
1  #include <pthread.h>
2
3  void * simple(void *);
4
5  #define NUM_THREADS 2
6  pthread_t tid[NUM_THREADS];      /* array de IDs de
7  threads*/
8
9  int bignum = 0;
```

```
10 pthread_spinlock_t bignum_lock;
11
12 main( int argc, char *argv[] )
13 {
14   int i, ret;
15
16   /* Inicializa o spin-lock */
17   pthread_spin_init(&bignum_lock, PTHREAD_PROCESS_PRIVATE);
18
19   /* Cria 2 threads simples */
20   for (i=0; i<NUM_THREADS; i++) {
21     pthread_create(&tid[i], NULL, simple, NULL);
22   }
23
24   /* Espera as threads terminarem */
25   for ( i = 0; i < NUM_THREADS; i++)
26     pthread_join(tid[i], NULL);
27
28   /* Destrói o spin-lock */
29   pthread_spin_destroy(&bignum_lock);
30 }
31
32
33 /* Thread simples */
34 void * simple(void * parm)
35 {
36   int i;
37   for(i=0;i<10000;i++) {
38      pthread_spin_lock(&bignum_lock); /* acquire - entra em
39 espera ocupada se não conseguir */
40        bignum++;  /* Região crítica */
41      pthread_spin_unlock(&bignum_lock); /* relinquish */
42   }
   }
```

Figura 55 – ***Utilização de spin-lock***

12.1.3. Semáforos

É possível utilizar semáforos em C através da inclusão da biblioteca semaphore.h. Os mutex também podem ser implementados através de semáforos binários, e assim realizar a exclusão mútua. Entretanto, os semáforos também possuem outras utilidades além desta, como por exemplo o controle de buffers em aplicações cliente/servidor.

Uma variável semáforo deve ser do tipo `sem_t`, e deve ser inicializada antes de ser utilizada. As principais abstrações para a manipulação de semáforos são:

Criação e destruição de semáforos:

`sem_init(semaphore, valorInicial, pshared)` : Inicializa uma variável semáforo. O parâmetro *pshared* indica se o semáforo vai ser compartilhado entre processos;

`sem_destroy(semaphore)` : Destrói uma variável semáforo.

Operações UP/DOWN:

`sem_wait(semaphore)` : Bloqueia se o seu valor for zero e decrementa o seu valor, caso contrário (operação wait/DOWN);

`sem_trywait(semaphore)` : Tenta obter acesso ao semáforo – testa se o seu valor é zero, e caso não seja, realiza a operação sem_wait() (não-bloqueante);

`sem_post(sempahore)` : Se o valor do semáforo for zero, acorda quem está bloqueado nele. Caso ninguém mais esteja bloqueado nele, incrementa o seu valor (operação signal/UP).

Ilustraremos agora, como semáforos binários (só podem assumir valor 0 e 1) podem ser utilizados como mutex. Para isso, o mesmo exemplo da figura 54 é mostrado na figura 56, agora utilizando semáforos. Duas threads que compartilham uma mesma variável (bignum) são criadas, e o acesso a esta variável (região crítica) é gerenciado através do semáforo bignum_binSem. Quando uma thread tenta obter acesso a um semáforo que já está travado(tem o seu valor 0), esta thread é então bloqueada.

```
1  #include <pthread.h>
2  #inluce <semaphore.h>
3
4  void * simple(void *);
5
6  #define NUM_THREADS 2
7  pthread_t tid[NUM_THREADS];      /* array de IDs de
8  threads*/
9
10 int bignum = 0;
11 sem_t bignum_binSem;
12
13 main( int argc, char *argv[] )
14 {
15   int i, ret;
16
17   /* Inicializa o semáforo com valor inicial 1 */
18   sem_init(&bignum_binSem, 1, 0);
19
20   /* Cria 2 threads simples */
21   for (i=0; i<NUM_THREADS; i++) {
22     pthread_create(&tid[i], NULL, simple, NULL);
23   }
24
25   /* Espera as threads terminarem */
26   for ( i = 0; i < NUM_THREADS; i++)
27     pthread_join(tid[i], NULL);
28
29   /* Destrói o semáforo */
30   sem_destroy(&bignum_binSem)
31 }
32
```

```
33
34 /* Thread simples */
35 void * simple(void * parm)
36 {
37   int i;
38   for(i=0;i<10000;i++) {
39      sem_wait(&bignum_binSem); /* acquire - entra em espera
40 ocupada se não conseguir */
41        bignum++;  /* Região crítica */
42      sem_post(&bignum_binSem); /* relinquish */
43   }
   }
```

Figura 56 – ***Utilizando mutex através de semáforo binário***

12.2. Eventos

C suporta o conceito de eventos através das variáveis de condição (condition variables), da biblioteca Pthreads. As variáveis de condição permitem a sincronização entre threads baseada em valores de variáveis. Variáveis de condição devem ser do tipo `pthread_cond_t`, e devem ser inicializadas antes de serem utilizadas. As principais abstrações usadas para manipular variáveis de condição são:

Criação e destruição de cariáveis de condição:

`pthread_cond_init(condition, attr)` : Inicializa uma variável de condição. O parâmetro *attr* é opcional;

`pthread_spin_destroy(condition)` : Destrói uma variável de condição.

Esperar e sinalizar variáveis de condição:

`pthread_cond_wait(condition, mutex)` : Bloqueia a thread que chamou esse método até que a condição especificada seja atingida (evento). O mutex deve estar travado (com valor 0) quando esta rotina é chamada. A rotina tratará então de liberar o mutex para que a variável possa ser manipulada por outras threads, e logo após põe o processo para dormir esperando até que a condição seja atingida.

`pthread_cond_signal(condition)` : Utilizada para sinalizar (ou acordar) outra thread que está esperando nesta variável condição.

`pthread_cond_broadcast(condition)` : Deve ser utilizada ao invés de `pthread_cond_signal()` se mais de uma thread estiver no estado bloqueado nesta condição.

```
1 /* Código da thread que espera pela condição */
2 Pthread_mutex_lock(&flagMutex );
3 if (flag == 0)
4    pthread_cond_wait(&condition , &flagMutex); /*aguarda
5 condição*/
6 else
```

```
7     pthread mutex unlock(&flagMutex);
8
9
10  /* Código da thread que sinaliza a condição */
11  pthread_mutex_lock(&flagMutex) ;
12  flag = 1 ;
13  pthread_mutex_unlock(&flagMutex) ;
    pthread_cond_signal(&condition) ; /*condição atingida*/
```

Figura 57 – ***Exemplo da utilização de eventos***

A figura 57 ilustra a utilização de eventos em C. O mutex *flagMutex* é utilizado para controlar o acesso a variável *flag*, e por este motivo ele é bloqueado na linha 1. Nas linhas 3 e 4, caso *flag* seja igual a zero, a thread será bloqueada na variável de condição *condition*, e o *flagMutex* será liberado pela rotina enquanto essa espera ocorre, para que a variável *flag* possa ser acessada e modificada por outras threads. Nas linhas 9-11, outra thread obtem acesso ao *flagMutex*, modifica o valor da variável *flag* e libera o mutex em seguida, e na linha 12 a variável de condição é sinalizada(a condição foi atingida/o evento ocorreu), o que faz com que as threads esperando nesta variável se acordem e continuem o seu processamento.

12.3. Mensagens

Mensagens podem ser enviadas/recebidas em C através da utilização da biblioteca MPI (Message Passing Interface). Essa biblioteca é muito extensa, e muitos livros são dedicados somente a escrita de programas utilizando-a. Como o conjunto de funções é bastante extenso (mais de 125), apresentaremos apenas as 6 abstrações principais, com as quais é possível desenvolver aplicações simples:

- **`MPI_Init :`** inicialização para o ambiente MPI;
- **`MPI_Comm_size :`** retorna o número de processos;
- **`MPI_Comm_rank :`** retorna o "rank" (índice, identificador) do processo;
- **`MPI_Send :`** envia uma mensagem;
- **`MPI_Recv :`** recebe uma mensagem;
- **`MPI_Finalize :`** sai do ambiente MPI.

No programa “Hello”, exibido na figura 58, cada processo executa uma cópia do programa. Todos os processos executam a chamada `MPI_Init` (linha 12) para a inicialização do ambiente de troca de mensagens. A pergunta "Quantos processos existem?" é respondida pelo comando `MPI_Comm_size` (linha 13) que retorna o número de processos (size) envolvidos. E a pergunta "Quem sou eu entre todos os processos?" é respondida pelo comando `MPI_Comm_rank` (linha 14) que retorna o número inteiro (rank), entre 0 e size-1, que identifica o processo que fez a chamada. Então um processo (aquele com rank 0) envia mensagens em um laço (`MPI_Send` – linha 20), colocando como argumento de destino o rank de cada um dos demais processos de forma que cada um deles receba uma mensagem. Este processo funciona como um processo Mestre. Os demais processos (rank maior ou igual a 1), recebem uma mensagem (`MPI_Recv` – linha 23), e funcionam como processos trabalhadores. Todos os processos imprimem a mensagem e abandonam o ambiente MPI (`MPI_Finalize` – linha 27).

```
1  #include <stddef.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include "mpi.h"
6  main(int argc, char **argv
7  {
8    char message[20];
9    int  i, rank, size, type = 99;
10   MPI_Status status;
11
12   MPI_Init(&argc, &argv);
13   MPI_Comm_size(MPI_COMM_WORLD, &size);
14   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16   if (rank == 0)
17   {
18     strcpy(message, "Hello, world");
19     for (i = 1; i < size; i++)
20        MPI_Send(message, 13, MPI_CHAR, i, type,
21 MPI_COMM_WORLD);
22   }
23   else
24      MPI_Recv(message, 20, MPI_CHAR, 0, type,
25 MPI_COMM_WORLD, &status);
26
27   printf( "Message from process = %d : %.13s\n",
28 rank,message);
     MPI_Finalize();
     }
```

Figura 58 – ***Envio e recebimento de mensagens com MPI***

12.4. RPC

C possui uma biblioteca RPC (rpc.h), onde é possível através dela implementar chamadas remotas a procedimento. Por possuir sintaxe extensa e complexa, ilustraremos os comandos básicos através do exemplo simples exibido nas figuras 59 e 60, onde uma aplicação cliente faz uma chamada remota a procedimento, e a aplicação servidor retorna uma string contendo “Hello World!”, para a aplicação que fez a chamada remota, que imprime o resultado na tela.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <ctype.h>
4  #include <rpc.h>
5
6  char* server()
7  {
8       char *result, helloWorld[20];
9
10     sprintf(helloWorld, "Hello World!");
11     result = helloWorld;
12
```

```
13      return result;
14  }
15
16
17  int main(void)
18  {
19      /* servicenumber=1, procnumber=1, version=1*/
20
21
22  registerrpc(1,1,1,server,xdr_wrapstring,xdr_wrapstring);
23
24      /* xdr_wrapstring eh um comando que transfere arrays de
25         caracteres */
26
27
28      svc_run(); /* roda o servidor RPC */
29      fprintf(stderr,"svc_run() nunca deve retornar!\n");
30      return 1;
    }
```

Figura 59 – *Stub RPC servidor - rpcserver.c*

Na figura 59 é mostrado o código da aplicação servidor, que vai hospedar o procedimento a ser acessado remotamente. Na linha 21, a função `server()` é registrada através da rotina `registerrpc()` com o numero de serviço 1 (utilizado pelos clientes para identificar o procedimento desejado). Na linha 27 o servidor RPC é colocado para rodar através da chamada `svc_run()`, onde permanece para atender as requisições dos clientes.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/types.h>
4   #include <sys/stat.h>
5   #include <fcntl.h>
6   #include <rpc.h>
7   #include <errno.h>
8
9   int main(int argc, char **argv)
10  {
11      int   result;
12      static char *output;
13
14      if(argc != 2)
15      {
16          fprintf(stderr,"Usage: %s
17  serverAddress\n",argv[0]);
18          exit(0);
19      }
20
21      /* host=argv[1], servicenumber=1, procnumber=1,
22  version=1
23         args=NULL */
24
25      result=callrpc(argv[1],1,1,1,xdr_wrapstring, NULL,
26                     xdr_wrapstring, &output);
```

```
27
28      /* xdr_wrapstring eh um comando que transfere arrays de
29         caracteres */
30
31       fprintf(stdout,"%s", output);
         return(1);
    }
```

Figura 60 – *Stub RPC cliente - rpcclient.c*

Na figura 60, a aplicação cliente faz uma chamada remota ao método `server()` do servidor, que tem como número de serviço 1, através da chamada `callrpc()` (linha 23), e salva o resultado na variável output. Ao final, o resultado é impresso na saída padrão (linha 29), imprimindo a string "Hello World!" que foi retornada pelo servidor.

13. Decisões de Projeto

Foi definido pelo grupo que o tabuleiro possuiria uma borda, além das casas que podem ser jogadas pelos jogadores. Essa decisão facilitou a realização de certas operações sobre o tabuleiro, como o acesso às casas adjacentes: não foi mais necessário verificar se a casa estava em um dos extremos do tabuleiro para evitar acessar um endereço de memória inválido.

Também foi adotado pelo grupo a abordagem de permitir que os jogadores saibam o valor de uma casa, estando ela visível (já foi jogada) ou não. Isso permite que diversos tipos de inteligência sejam projetados para o jogo, inclusive um jogador deus – onisciente, tudo sabe, sempre acerta as minas.

O grupo decidiu tornar o jogo portável. Dessa maneira, o código pode ser compilado tanto em ambientes Windows quanto Linux, sem perda da jogabilidade. Para isso, foram usadas diretivas de pré-compilação, que compilam o código referente ao ambiente no qual o compilador está sendo executado. As operações que necessitaram de comandos multiplataforma foram as de interface, como a limpeza da tela, o travamento do console, e o desenho do tabuleiro.

A inteligência adotada para as jogadas do computador mostrou-se bastante funcional. A estratégia utilizada foi de buscar pela casa visível que possui maior probabilidade de ter bombas adjacentes. O computador joga em uma casa adjacente a essa casa numerada que possui chances reais de ter bomba, analisando as casas vizinhas. Destacamos o fato de que o computador tem a mesma visão do tabuleiro que o jogador humano.

14. Conclusão

A implementação do Campo Minado com a linguagem C mostrou-se inicialmente um trabalho árduo em se tratando da grande responsabilidade atribuída ao programador do correto acesso à memória e, principalmente, da necessidade do uso de ponteiros em alguns trechos do jogo (já que, por exemplo, as casas eram tratadas através dos mesmos). Isto foi considerado uma dificuldade pois é necessária certa experiência para manipular dados corretamente utilizando ponteiros. Qualquer manipulação errada, e o código não funciona, ficando muito difícil encontrar o problema.

Porém, por possuir uma sintaxe com qual o grupo estava, de certa forma, familiarizado (pela similaridade em alguns aspectos com a da linguagem Pascal e de Java), pela liberdade que C fornece ao programador e pelo rápido tempo de execução de seus programas, o resultado do jogo foi bastante satisfatório.

15. Referências

[1] http://en.wikipedia.org/wiki/C_%28programming_language%29
[2] http://en.wikipedia.org/wiki/ANSI_C
[3] http://en.wikipedia.org/wiki/Programming_paradigm
[4] http://publications.gbdirect.co.uk/c_book/
[5] http://www.cs.cf.ac.uk/Dave/C/
[6] http://www.cs.cornell.edu/courses/cs414/2001SP/tutorials/cforjava.htm
[7] http://csdir.org/tutorials/ansi-c-tutorial/
[8] http://en.wikipedia.org/wiki/GNU_Compiler_Collection
[9] http://en.wikipedia.org/wiki/Programming_language_implementation
[10] http://irc.essex.ac.uk/www.iota-six.co.uk/c/
[11] http://www.java2s.com/Tutorial/C/0140__Array/Passingarraysandindividualarrayelementstofunctions.htm
[12] http://en.wikipedia.org/wiki/C_syntax
[13] http://www.cppreference.com/
[14] http://en.wikipedia.org/wiki/C_variable_types_and_declarations
[15] http://en.wikipedia.org/wiki/Malloc
[16] http://home.netcom.com/~tjensen/ptr/
[17] http://www.cplusplus.com/reference/clibrary/
[18] http://www.newty.de/fpt/
[19] http://www.cs.jcu.edu.au/Subjects/cp2003/1998/ch7Type/typeSystems.html
[20] http://www.c-faq.com/
[21] http://www.comeaucomputing.com/
[22] http://www.acm.uiuc.edu/webmonkeys/book/c_guide/
[23] http://thewizardstower.org/thelibrary/programming/polyc.html
[24] http://www.cs.jcu.edu.au/Subjects/cp2003/1998/ch7Type/typeSystems.html
[25] https://computing.llnl.gov/tutorials/pthreads
[26] http://cognitus.net/html/howto/pthreadSemiFAQ_9.html
[27] http://www.jbox.dk/sanos/source/lib/pthread/spinlock.c.html
[28] http://www.opengroup.org/onlinepubs/000095399/basedefs/semaphore.h.html
[29] http://www.inf.puc-rio.br/~alvim/MPI/ProgramasMPI.htm
[30] http://gcc.gnu.org/onlinedocs/gcc-4.3.0//cpp/Header-Files.html
[31] http://computerprogramming.suite101.com/article.cfm/c_header_files
[32] http://www.gnu.org/software/libtool/manual/libc/Header-Files.html
[33] http://gcc.gnu.org/onlinedocs/gcc-4.3.0//cpp/Header-Files.html
[34] http://www.cim.mcgill.ca/~franco/OpSys-304-427/messages/node119.html
[35] http://www2.cs.uregina.ca/~hamilton/courses/430/notes/rpc.html
[36] http://www.cprogramming.com/
[37] http://www.cs.grinnell.edu/~stone/courses/languages/C-syntax.xhtml
[38] http://www.softlab.ntua.gr/~nickie/