CS 241: programação do sistema

Introc

Se você quer ensinar sistemas, não instigue os programadores, classifique os problemas e faça os PRs. Em vez disso, ensine-os a ansiar pelo vasto e infinito C. - **Antoine de Saint-Exupéry (Com edições de Bhuvy)**
C é a linguagem de programação de fato para fazer uma séria programação séria do sistema. Por quê? A maioria dos kernels é escrita em grande parte em C. O kernel do Linux (Love # ref-Love) e o kernel XNU Inc. (# ref-xnukernel) do qual o Mac OS X é baseado. O Kernel do Windows usa C ++, mas fazer a programação do sistema é muito mais difícil no Windows que o UNIX para programadores de sistemas iniciantes. A maioria de vocês tem alguma experiência com C ++, mas C é uma besta completamente diferente. Você não tem boas abstrações como classes e RAII para limpar a memória. Você vai ter que fazer isso sozinho. C dá-lhe muito mais uma oportunidade para se atirar no pé, mas permite que você faça as coisas em um nível de grão muito mais fino.

História do C

C foi desenvolvido por Dennis Ritchie e Ken Thompson no Bell Labs em 1973 (Ritchie # ref-Ritchie: 1993: DCL: 155360.155580). Naquela época, tínhamos gemas de linguagens de programação como Fortran, ALGOL e LISP. O objetivo de C era duas vezes. Um, para atingir os computadores mais populares da época, como o PDP-7. Dois, tente remover algumas das construções de nível inferior, como gerenciar registros, programar assembly para saltos e, em vez disso, criar uma linguagem que tenha o poder de expressar programas processualmente (em oposição a matematicamente como lisp) com código mais legível e ainda ter a capacidade para interagir com o sistema operacional. Parecia um feito difícil. Inicialmente, era usado apenas internamente na Bell Labs junto com o sistema operacional UNIX.
A primeira padronização “real” é com o livro de Brian Kernighan e Dennis Ritchie (Kernighan e Ritchie # ref-kernighan1988c). Ainda é amplamente considerado hoje como o único conjunto portátil de instruções C. O livro K \ & R é conhecido como o padrão de fato para o aprendizado C. Havia padrões diferentes de C de ANSI para ISO após os guias Unix. Aquela em que estaremos nos concentrando principalmente é a biblioteca POSIX C. Agora, para tirar o elefante da sala, o kernel do Linux não é totalmente compatível com POSIX. Principalmente, é porque eles não queriam pagar a taxa pela conformidade, mas também não quer ser completamente compatível com um monte de padrões diferentes, porque

então tem que resultar em custos crescentes de desenvolvimento para manter a conformidade.
Avanço rápido, no entanto, muitos anos, e estamos no atual padrão C apresentado pela ISO: C11. Nem todo o código que nós, nessa classe, estaremos nesse formato. Vamos tentar usar o C99 como padrão que a maioria dos computadores reconhece. Falaremos sobre alguns recursos off-hand como o getline porque eles são muito usados com a biblioteca GNU-C. Começaremos fornecendo uma visão geral abrangente e decente do idioma com as instalações de emparelhamento.

Recursos

- Rápido. Há muito pouco que separa você e o sistema.
- Simples. C e sua biblioteca padrão representam um conjunto simples de funções portáteis.
- Gerenciamento de Memória. C deixa você gerenciar sua memória. Isso também pode te morder se você tiver erros de memória.
- Está em toda parte. Praticamente todo computador que não está embutido tem alguma forma de interagir com o C. A biblioteca padrão também está em todo lugar. C resistiu ao teste do tempo como uma linguagem popular, e não parece que está indo a lugar algum.

Introdução ao curso Crash para C

A única maneira de começar a aprender C é começar com o hello world. De acordo com o exemplo original que Kernighan e Ritchie propuseram quando, o mundo hello não mudou muito.
#include <stdio.h> int main (void) { printf ("Hello World \n "); return 0 ; }

1. A diretiva #include usa o arquivo stdio.h (que significa **st** ent **d** orput and oputput) localizado em algum lugar do sistema operacional, copia o texto e o substitui onde o #include estava.
2. O int main(void) é uma declaração de função. A primeira palavra int informa ao compilador qual é o tipo de retorno da função. A parte antes do parêntese (main) é o nome da função. Em C, duas funções não podem ter o mesmo nome em um único programa compilado, as bibliotecas compartilhadas são um assunto diferente. Então, o que vem depois é a lista de parâmetros. Quando damos a lista de parâmetros para funções regulares (void) isso significa que o compilador deve errar se a função for chamada com algum argumento. Para funções regulares ter uma declaração como void func() significa que você tem permissão para chamar a função como func(1, 2, 3) porque não há delimitador. No caso de main , é uma função especial. Há muitas maneiras de declarar main mas as que você estará familiarizado são int main(void) , int main() e int main(int argc, char *argv[]) .
3. printf("Hello World"); é o que chamamos de chamada de função. printf é definido como parte de stdio.h . A função foi compilada e mora em outro lugar em nossa máquina. Tudo o que precisamos fazer é incluir o cabeçalho e chamar a função com os parâmetros apropriados (uma string literal "Hello World"). Se você não tiver a

nova linha, o buffer não será liberado. É por convenção que o IO em buffer não é liberado até uma nova linha.
4. return 0; . main tem que retornar um inteiro. Por convenção, return 0 significa sucesso e qualquer outra coisa significa falha.

$ gcc main.c -o main $ ./main Hello World $

1. gcc é a abreviação de GNU-Compiler-Collection, que possui uma série de compiladores prontos para uso. O compilador infere da extensão que você está tentando compilar um arquivo .c
2. ./main diz ao seu shell para executar o programa no diretório atual chamado main. O programa então mostra o mundo

Pré-processador

Qual é o pré-processador? O pré-processamento é uma operação que o compilador executa **antes de** realmente compilar o programa. É um comando copiar e colar. Significado a seguinte substituição é executada.
#define MAX_LENGTH 10 char buffer [MAX_LENGTH] // After char buffer [10]
Há efeitos colaterais para o pré-processador embora. Um problema é que o pré-processador precisa ser capaz de tokenizar corretamente, o que significa que tentar redefinir as partes internas da linguagem C com um pré-processador pode ser impossível. Outro problema é que eles não podem ser aninhados infinitamente - existe uma profundidade ilimitada onde eles precisam parar. Macros também são apenas substituições de texto simples. Por exemplo, observe o que pode acontecer se tivermos uma macro que realiza uma modificação inline.
#define min(a,b) ((a)<(b) ? (a) : (b)) int main () { int x = 4 ; if (min (x ++ , 5)) printf ("%d is six" , x); return 0 ; }
Macros são simples substituição de texto, então o exemplo acima se expande para x++ < 100 ? x++ : 100 x++ < 100 ? x++ : 100 (parênteses omitidos para maior clareza). Agora, para este caso, é opaco o que é impresso, mas será 6. Considere também o caso de borda quando a precedência do operador entra em jogo.
#define min(a,b) a < b ? a : b int x = 99 ; int r = 10 + min (99 , 100); // r is 100! // This is what it is expanded to int r = 10 + 99 < 100 ? 99 : 100 // Which means int r = (10 + 99) < 100 ? 99 : 100
Você também pode ter problemas lógicos com a flexibilidade de certos parâmetros. Uma fonte comum de confusão é com matrizes estáticas e o operador sizeof .
#define ARRAY_LENGTH(A) (sizeof((A)) / sizeof((A)[0])) int static_array [10]; // ARRAY_LENGTH(static_array) = 10 int * dynamic_array = malloc (10); // ARRAY_LENGTH(dynamic_array) = 2 or 1
O que há de errado com a macro? Bem, isso funciona se tivermos uma matriz estática como a primeira matriz porque sizeof uma matriz estática retorna o número de bytes que a matriz ocupa e dividi-la pelo sizeof(an_element) forneceria o número de entradas. Mas se usarmos um ponteiro para uma parte da memória, tomar o tamanho do ponteiro e dividi-lo pelo tamanho da primeira entrada nem sempre nos dará o tamanho da matriz.

Extra: inclui e condicionais

O outro pré-processador inclui a diretiva #include e os condicionais. A diretiva include é explicada pelo exemplo.
// foo.h int bar ();
// foo.c unpreprocessed #include "foo.h" int bar () { }
Após o pré-processamento, o compilador vê isso
// foo.c unpreprocessed int bar (); int bar () { }
A outra coisa que temos é condicionais. Se uma macro é definida ou igual a zeros, essa ramificação não é
int main () { #ifdef __GNUC__ return 1 ; #else return 0 ; #endif }
Usando o gcc seu compilador veria isso
int main () { return 1 ; }
Usando clang seu compilador veria isso
int main () { return 0 ; }

Instalações de idioma

Palavras-chave

C tem uma variedade de palavras-chave. Aqui estão algumas construções que você deve conhecer brevemente a partir de C99.

1. break é uma palavra-chave usada em instruções case ou declarações em loop. Quando usado em uma declaração de caso, o programa pula para o final do bloco.
   switch (1) { case 1 : /* Goes to this switch */ puts ("1"); break ; /* Jumps to the end of the block */ case 2 : /* Ignores this program */ puts ("2"); break ; } /* Continues here */
   No contexto de um loop, ele rompe o loop mais interno. O loop pode ser uma construção for , while ou do-while
   while (1) { while (2) { break ; /* Breaks out of while(2) */ } /* Jumps here */ break ; /* Breaks out of while(1) */ } /* Continues here */
2. const é uma construção em nível de linguagem que informa ao compilador que esses dados não devem ser modificados. Se alguém tentar alterar uma variável const, o programa nem será compilado. const funciona um pouco diferente quando colocado antes do tipo, o compilador inverte o primeiro tipo e const. Em seguida, o compilador usa uma regra de associatividade à esquerda. O que significa que o que resta do ponteiro é constante. Isso é conhecido como correção de const.
   const int i = 0 ; // Same as "int const i = 0" char * str = ...; // Mutable pointer to a mutable string const char * const_str = ...; // Mutable pointer to a constant string char const * const_str2 = ...; // Same as above const char * const const_ptr_str = ...; // Constant pointer to a constant string
   Mas, é importante saber que isso é apenas uma restrição imposta pelo compilador. Existem maneiras de contornar isso e o programa rodará bem com comportamento definido. Na programação de sistemas, o único tipo de memória que você não pode gravar é a memória protegida contra gravação do sistema.

const int i = 0 ; // Same as "int const i = 0" (* ((int *) & i)) = 1 ; // i == 1 now const char * ptr = "hi" ; * ptr = '\0' ; // Will cause a Segmentation Violation

3. continue é uma instrução de fluxo de controle que existe apenas em construções de loop. Continue irá pular o resto do corpo do loop e definir o contador do programa de volta ao início do loop antes.
   int i = 10 ; while (i --) { if (1) continue ; /* This gets triggered */ * ((int *) NULL) = 0 ; } /* Then reaches the end of the while loop */
4. do {} while(); é outro loop constrói. Esses loops executam o corpo e, em seguida, verificam a condição na parte inferior do loop. Se a condição for zero, o corpo do loop não será executado e o restante do programa será executado. Caso contrário, o corpo do loop é executado.
   int i = 1 ; do { printf ("%d \n " , i --); } while (i > 10) /* Only executed once */
5. enum é declarar uma enumeração. Uma enumeração é um tipo que pode assumir muitos valores finitos. Se você tiver um enum e não especificar nenhum numérico, o compilador c quando gerar um número exclusivo para esse enum (dentro do contexto do enum atual) e usar isso para comparações. Para declarar uma instância de um enum, você deve dizer enum <type> varname . O benefício adicional disso é que C pode digitar essas expressões para se certificar de que você está apenas comparando tipos semelhantes.
   enum day { monday , tuesday , wednesday , thursday , friday , saturday , sunday }; void process_day (enum day foo) { switch (foo) { case monday : printf ("Go home! \n "); break ; // ... } }
   É completamente possível atribuir valores enum para serem diferentes ou iguais. Apenas não confie no compilador para numeração consistente. Se você for usar essa abstração, tente não quebrá-la.
   enum day { monday = 0 , tuesday = 0 , wednesday = 0 , thursday = 1 , friday = 10 , saturday = 10 , sunday = 0 }; void process_day (enum day foo) { switch (foo) { case monday : printf ("Go home! \n "); break ; // ... } }
6. extern é uma palavra-chave especial que diz ao compilador que a variável pode ser definida em outro arquivo objeto ou biblioteca, então o compilador não lança um erro quando a variável não é definida ou se a variável é definida duas vezes porque o primeiro arquivo será realmente referenciando a variável no outro arquivo.
   // file1.c extern int panic ; void foo () { if (panic) { printf ("NONONONONO"); } else { printf ("This is fine"); } } //file2.c int panic = 1 ;
7. for é uma palavra-chave que permite iterar com uma condição de inicialização, uma invariante de loop e uma condição de atualização. Isso deve ser um substituto para o loop while
   for (initialization ; check ; update) { //... } // Typically int i ; for (i = 0 ; i < 10 ; i ++) { //... }
   A partir do padrão C89, você não pode declarar variáveis dentro do loop for . Isso ocorre porque houve um desacordo no padrão de como as regras de escopo de uma variável definida no loop funcionariam. Desde então, foi resolvido com padrões mais recentes, para que as pessoas possam usar o loop for que conhecem e amam hoje
   for (int i = 0 ; i < 10 ; ++ i) { }
   A ordem de avaliação de um loop for é a seguinte
   1. Execute a condição de inicialização.

    2. Verifique o invariante. Se for falso, termine o loop e execute a próxima instrução. Se for verdade, continue com o corpo do loop.
    3. Execute o corpo do loop.
    4. Execute a condição de atualização.
    5. Pule para verificar a etapa invariante.
8. goto é uma palavra-chave que permite fazer saltos condicionais. Não use goto em seus programas. A razão disso é que torna seu código infinitamente mais difícil de entender quando ligado a várias cadeias. É bom usar em alguns contextos. A palavra-chave é normalmente usada em contextos de kernel quando adicionar outro quadro de pilha para limpeza não é uma boa ideia. O exemplo canônico de limpeza do kernel é como abaixo.
   void setup (void) { Doe * deer ; Ray * drop ; Mi * myself ; if (! setupdoe (deer)) { goto finish ; } if (! setupray (drop)) { goto cleanupdoe ; } if (! setupmi (myself)) { goto cleanupray ; } perform_action (deer , drop , myself); cleanupray: cleanup (drop); cleanupdoe: cleanup (deer); finish: return ; }
9. if else else-if são palavras-chave do fluxo de controle. Existem algumas maneiras de usá-los (1) Um nulo if (2) An if com um else (3) um if com um else-if (4) um if com um else if e else. As instruções são sempre executadas do if para o else. Se alguma das condições intermediárias for verdadeira, o bloco if executa essa ação e vai para o final desse bloco.
   // (1) if (connect (...)) return - 1 ; // (2) if (connect (...)) { exit (- 1); } else { printf ("Connected!"); } // (3) if (connect (...)) { exit (- 1); } else if (bind (..)) { exit (- 2); } // (1) if (connect (...)) { exit (- 1); } else if (bind (..)) { exit (- 2); } else { printf ("Successfully bound!"); }
10. inline é uma palavra-chave do compilador que informa ao compilador que não há problema em criar uma nova função no assembly. Em vez disso, a compilação é sugerida em substituir o corpo da função diretamente na função de chamada. Isso nem sempre é recomendado explicitamente, já que o compilador geralmente é inteligente o suficiente para saber quando inline uma função para você.
    inline int max (int a , int b) { return a < b ? a : b ; } int main () { printf ("Max %d" , max (a , b)); // printf("Max %d", a < b ? a : b); }
11. restrict é uma palavra-chave que informa ao compilador que essa região de memória específica não deve se sobrepor a todas as outras regiões da memória. O caso de uso para isso é informar aos usuários do programa que o comportamento é indefinido se as regiões de memória se sobrepuserem.
    memcpy (void * restrict dest , const void * restrict src , size_t bytes); void add_array (int * a , int * restrict c) { * a += * c ; } int * a = malloc (3 * sizeof (* a)); * a = 1 ; * a = 2 ; * a = 3 ; add_array (a + 1 , a) // Well defined add_array (a , a) // Undefined
12. return é um operador de fluxo de controle que sai da função atual. Se a função for void , simplesmente sai das funções. Caso contrário, outro parâmetro segue como o valor de retorno.
    void process () { if (connect (...)) { return - 1 ; } else if (bind (...)) { return - 2 } return 0 ; }
13. signed é um modificador que raramente é usado, mas força um tipo a ser assinado em vez de não assinado. A razão pela qual isso é tão raramente usado é porque os tipos são assinados por padrão e precisam ter o modificador unsigned para torná-los

não assinados, mas pode ser útil nos casos em que você deseja que o compilador defina um tipo assinado como padrão.

int count_bits_and_sign (signed representation) { //... }

14. sizeof é um operador que é avaliado em tempo de compilação, que avalia o número de bytes que a expressão contém. Isso significa que quando o compilador digita o tipo, o código a seguir é alterado.

    char a = 0 ; printf ("%zu" , sizeof (a ++));

    char a = 0 ; printf ("%zu" , 1);

    Em seguida, o compilador pode operar mais. Uma nota que você deve ter uma definição completa do tipo em tempo de compilação ou então você pode obter um erro estranho. Considere o seguinte

    // file.c struct person ; printf ("%zu" , sizeof (person)); // file2.c struct person { // Declarations }

    Este código não irá compilar porque sizeof não é capaz de compilar file.c sem conhecer a declaração completa da person struct. Normalmente, é por isso que colocamos a declaração completa em um arquivo de cabeçalho ou abstraímos a criação e a interação para que os usuários não possam acessar os componentes internos de nossa estrutura. Além disso, se o compilador souber o comprimento total de um objeto de matriz, ele usará isso na expressão em vez de decaí-lo para um ponteiro.

    char str1 [] = "will be 11" ; char * str2 = "will be 8" ; sizeof (str1) //11 because it is an array sizeof (str2) //8 because it is a pointer

    Tenha cuidado, usando sizeof para o comprimento de uma string!

15. static é um especificador de tipo com três significados.
    1. Quando usado com uma variável global ou declaração de função, significa que o escopo da variável ou da função é limitado apenas ao arquivo.
    2. Quando usada com uma variável de função, ela declara que a variável tem alocação estática - o que significa que a variável é alocada uma vez no início do programa, não toda vez que o programa é executado.
16. static int i = 0 ; static int _perform_calculation (void) { // ... } char * print_time (void) { static char buffer [200]; // Shared every time a function is called // ... }
17. struct é uma palavra-chave que permite emparelhar vários tipos em uma nova estrutura. Estruturas são regiões contíguas de memória que podem acessar elementos específicos de cada memória como se fossem variáveis separadas.

    struct hostname { const char * port ; const char * name ; const char * resource ; }; // You need the semicolon at the end // Assign each individually struct hostname facebook ; facebook . port = "80" ; facebook . name = "www.google.com" ; facebook . resource = "/" ; // You can use static initialization in later versions of c struct hostname google = { "80" , "www.google.com" , "/" };
18. switch case default Switches são essencialmente instruções de salto glorificadas. Significa que você toma um byte ou um inteiro e o fluxo de controle do programa salta para esse local.

    switch (/* char or int */) { case INT1 : puts ("1"); case INT2 : puts ("2"); case INT3 : puts ("3"); }

    Se dermos um valor de 2 então

    switch (2) { case 1 : puts ("1"); /* Doesn't run this */ case 2 : puts ("2"); /* Runs

this */ case 3 : puts ("3"); /* Also runs this */ }

Um dos exemplos mais famosos disso é o dispositivo de Duff, que permite o desenrolar dos circuitos. Você não precisa entender este código para os propósitos desta classe, mas é divertido de se olhar (Duff, # ref-duff).

send (to , from , count) register short * to , * from ; register count ; { register n = (count + 7) / 8 ; switch (count % 8){ case 0 : do { * to = * from ++ ; case 7 : * to = * from ++ ; case 6 : * to = * from ++ ; case 5 : * to = * from ++ ; case 4 : * to = * from ++ ; case 3 : * to = * from ++ ; case 2 : * to = * from ++ ; case 1 : * to = * from ++ ; } while (-- n > 0); } }

Esta parte do código destaca que as instruções switch são apenas instruções goto, e você pode colocar qualquer outra parte válida do código na outra extremidade de um caso de switch. Na maioria das vezes não faz sentido, algumas vezes faz muito sentido.

19. typedef declara um alias para um tipo. Freqüentemente usado com estruturas para reduzir a desordem visual de ter que escrever 'struct' como parte do tipo.

    typedef float real ; real gravity = 10 ; // Also typedef gives us an abstraction over the underlying type used. // In the future, we only need to change this typedef if we // wanted our physics library to use doubles instead of floats. typedef struct link link_t ; //With structs, include the keyword 'struct' as part of the original types

    Nesta classe, nós regularmente digitamos funções. Um typedef para uma função pode ser este por exemplo

    typedef int (* comparator)(void * , void *); int greater_than (void * a , void * b){ return a > b ; } comparator gt = greater_than ;

    Isto declara um comparador de tipo de função que aceita dois parâmetros void* e retorna um inteiro.
20. union é um novo tipo de especificador. Uma união é um pedaço de memória que um monte de variáveis ocupa. Ele é usado para manter a consistência e ter a flexibilidade de alternar entre os tipos sem manter as funções para acompanhar os bits. Considere um exemplo em que temos valores de pixel diferentes.

    union pixel { struct values { char red ; char blue ; char green ; char alpha ; } values ; uint32_t encoded ; }; // Ending semicolon needed union pixel a ; // When modifying or reading a . values . red ; a . values . blue = 0x0 ; // When writing to a file fprintf (picture , "%d" , a . encoded);
21. unsigned é um modificador de tipo que força o comportamento unsigned nas variáveis que eles modificam. Unsigned só pode estar em tipos int primitivos (como int e long). Há muito comportamento associado à aritmética não assinada, só sei que, na maioria das vezes, a menos que você precise fazer o bit bifting, provavelmente não será necessário.
22. void é uma palavra chave dobrada. Quando usado em termos de função ou definição de parâmetro, significa que não retorna nenhum valor ou não aceita nenhum parâmetro especificamente. O seguinte declara uma função que não aceita parâmetros e não retorna nada.

    void foo (void);

    O outro uso do void é quando você está definindo. Um void * pointer é apenas um endereço de memória. Ele é especificado como um tipo incompleto, o que significa que você não pode desreferencia, mas pode ser promovido a qualquer momento

para qualquer outro tipo. A aritmética do ponteiro com esse ponteiro é um comportamento indefinido.
int * array = void_ptr ; // No cast needed

23. volatile é uma palavra-chave do compilador. Isso significa que o compilador não deve otimizar seu valor. Considere a seguinte função simples.
int flag = 1 ; pass_flag (& flag); while (flag) { // Do things unrelated to flag }
O compilador pode, desde os internos do loop while não tem nada a ver com o sinalizador, otimizá-lo para o seguinte, mesmo que uma função pode alterar os dados.
while (1) { // Do things unrelated to flag }
Se você colocar a palavra-chave volatile, ela força o compilador a manter a variável e executar essa verificação. Isso é particularmente útil para casos em que você está executando programas multiprocesso ou multiencadeados para que possamos
24. while representa o loop while tradicional. Existe uma condição na parte superior do loop. Enquanto essa condição for avaliada como um valor diferente de zero, o corpo do loop será executado.

Tipos de dados C

Existem muitos tipos de dados em c. Todos eles, como você pode perceber, são um inteiro ou um número de ponto flutuante e tipos diferentes são variações disso.

1. char Representa exatamente um byte de dados. O número de bits em um byte pode variar. unsigned char signed char e unsigned char signed char significam exatamente a mesma coisa. Isso deve ser alinhado em um limite (o que significa que você não pode usar bits entre dois endereços). O resto dos tipos assumirá 8 bits em um byte.
2. short (short int) deve ter pelo menos dois bytes. Isso é alinhado em um limite de dois bytes, o que significa que o endereço deve ser divisível por dois.
3. int deve ter pelo menos dois bytes. Novamente alinhado a um limite de dois bytes (“ISO C Standard” # ref-ISON1124 P. 34). Na maioria das máquinas, isso será de 4 bytes.
4. long (long int) deve ter pelo menos quatro bytes, alinhados a um limite de quatro bytes. Em algumas máquinas, isso pode ter 8 bytes.
5. long long deve ter pelo menos oito bytes, alinhados a um limite de oito bytes.
6. float representa um número de ponto flutuante de precisão única IEEE-754 rigorosamente especificado pelo IEEE (“Padrão IEEE para Aritmética de Ponto Flutuante” # ref-4610935). Estes serão quatro bytes alinhados a um limite de quatro bytes na maioria das máquinas.
7. double representa um número de ponto flutuante de precisão dupla IEEE-754 especificado pelo mesmo padrão, que é alinhado ao limite de oito bytes mais próximo.

Operadores

Operadores são construções de linguagem em C que são definidas como parte da gramática da linguagem.

1. [] é o operador de índice. a[n] == (a + n)* onde n é um tipo numérico e a é um tipo de ponteiro.
2. -> é o operador de desreferencia da estrutura. Se você tiver um ponteiro para uma struct *p , poderá usar isso para acessar um de seus elementos. p->element .
3. . é o operador de referência da estrutura. Se você tiver um objeto na pilha a , poderá acessar um elemento a.element .
4. +/-a é o operador unário mais e menos. Eles mantêm ou negam o sinal, respectivamente, do inteiro ou do tipo flutuante embaixo.
5. *a é o operador de remoção de referência. Se você tiver um ponteiro *p , poderá usá-lo para acessar o elemento localizado nesse endereço de memória. Se você estiver lendo, o valor de retorno será o tamanho do tipo subjacente. Se você está escrevendo, o valor será escrito com um deslocamento.
6. &a é o operador de endereço. Isso leva o elemento an e retorna seu endereço.
7. ++ é o operador de incremento. Você pode pegar o prefixo ou o postfix, o que significa que a variável que está sendo incrementada pode ser antes ou depois do operador. a = 0; ++a === 1 a = 0; ++a === 1 e a = 1; a++ === 0 a = 1; a++ === 0 .
8. – é o operador de decréscimo. Mesma semântica que o operador de incremento, exceto com a diminuição do valor em um.
9. sizeof é o operador sizeof. Isso também é mencionado na seção de palavras-chave.
10. a <mop> b onde <mop>=+, -, *, %, / são os operadores binários matemáticos. Se os operandos são ambos tipos numéricos, então as operações são plus, menos, times, modulo e division respectivamente. Se o operando da esquerda é um ponteiro e o operando da direita é um tipo inteiro, somente mais ou menos pode ser usado e as regras para aritmética de ponteiro são invocadas.
11. >>/<< são os operadores de mudança de bit. O operando à direita deve ser um tipo inteiro cuja assinatura é ignorada, a menos que seja sinalizada em negativo, caso em que o comportamento é indefinido. O operador à esquerda decide muitas semânticas. Se ficarmos mudando, sempre haverá zeros à direita. Se estamos certos mudando, há alguns casos diferentes
    1. Se o operando à esquerda estiver assinado, o inteiro será sinal extendido. Isso significa que, se o número tiver o bit de sinal definido, qualquer deslocamento à direita introduzirá o lado esquerdo. Se o número não tiver o bit de sinal definido, qualquer deslocamento à direita introduzirá zeros à esquerda.
    2. Se o operando não estiver assinado, os zeros serão introduzidos à esquerda, de qualquer forma.
12. unsigned short uns = - 127 ; // 1111111110000001 short sig = 1 ; // 0000000000000001 uns << 2 ; // 1111111000000100 sig << 2 ; // 0000000000000100 uns >> 2 ; // 1111111111100000 sig >> 2 ; // 0000000000000000
13. <=/>= são o maior que igual a / menor que igual aos operadores. Eles fazem como o nome indica.
14. </> são os maiores que / menos que os operadores. Eles novamente fazem como o nome indica.

15. ==/= são iguais / não iguais aos operadores. Eles mais uma vez fazem como o nome indica.
16. && é o lógico e o operador. Se o primeiro operando for zero, o segundo não será avaliado e a expressão será avaliada como 0. Caso contrário, ele gerará um valor 1-0 do segundo operando.
17. || é o lógico ou operador. Se o primeiro operando não for zero, o segundo não será avaliado e a expressão será avaliada como
    1. Caso contrário, gera um valor 1-0 do segundo operando.
18. ! é o operador não lógico. Se o operando for zero, isso retornará 1. Caso contrário, retornará 0.
19. & Se um bit é definido nos dois operandos, é definido na saída. Caso contrário, não é.
20. | Se um bit for definido em um dos operandos, ele será definido na saída. Caso contrário, não é.
21. Se um bit for definido na entrada, ele não será definido na saída e vice-versa.
22. ?: é o operador ternário. Você coloca uma condição booleana antes de e se ela avalia como diferente de zero o elemento antes que os dois-pontos sejam retornados, caso contrário, o elemento após é. 1 ? a : b === a 1 ? a : b === a 0 ? a : b === b 0 ? a : b === b .
23. a, b é o operador de vírgula. a é avaliado e, em seguida, b é avaliado e b é retornado.

O C e o Linux

Então, até este ponto, cobrimos os fundamentos da linguagem com C. O que começa a divergir de todas as formas de C são as funções que usamos e como interagimos com o sistema operacional. Agora estaremos focando nossa atenção em C e na variedade POSIX de funções disponíveis para nós. Vamos falar sobre funções portáteis, por exemplo fwrite printf etc etc, mas estaremos avaliando os internos e examiná-los sob os modelos POSIX e mais especificamente LINUX. Existem várias coisas nessa filosofia que tornam o resto mais fácil de ser conhecido, então vamos colocar essas coisas aqui.

Tudo é um arquivo

Um mantra POSIX é que tudo é um arquivo. Embora isso tenha sido recentemente ultrapassado e, além disso, errado, é a convenção que ainda usamos hoje. O que significa POSIX é tudo o que é um arquivo é que tudo é um descritor de arquivo ou um inteiro. Por exemplo, aqui está um objeto de arquivo, um soquete de rede e um objeto de kernel
int file_fd = open (...); int network_fd = socket (...); int kernel_fd = epoll_create1 (...);
E as operações nesses objetos são feitas por meio de chamadas do sistema. Uma última coisa a notar antes de prosseguirmos é que os descritores de arquivos são meramente *ponteiros* . Imagine que cada um dos descritores de arquivo no exemplo realmente se refira a uma entrada em uma tabela de objetos que o sistema operacional seleciona e escolhe. Objetos podem ser alocados e desalocados, fechados e abertos, etc. O programa interage com esses objetos usando a API ou as chamadas do sistema especificadas

Chamadas do sistema

Antes de nos aprofundarmos nas funções comuns do C, precisamos saber o que é uma chamada do sistema. Se você é um estudante e completou o HW0, sinta-se à vontade para encobrir esta seção.
Uma chamada de sistema é uma operação que o kernel realiza em vez do programa. O sistema operacional prepara uma chamada do sistema e o kernel executa a chamada do sistema com o melhor de sua capacidade no espaço do kernel. No exemplo anterior, abrimos um objeto descritor de arquivo. Agora também podemos escrever alguns bytes para o objeto descritor de arquivo que representa um arquivo e o sistema operacional fará o melhor para obter os bytes gravados no disco.
write (file_fd , "Hello!" , 6);
Quando dizemos que o kernel tenta o seu melhor, a operação pode falhar por várias razões. O arquivo não é mais válido, o disco rígido falhou, o sistema foi interrompido etc etc. A maneira como você se comunica com o sistema externo é com as chamadas do sistema. A outra coisa a notar é que as chamadas do sistema são caras. Seu custo em termos de tempo e ciclos de CPU foi reduzido recentemente, mas tente usá-los com a maior moderação possível.

C Chamadas

Muitas chamadas C que discutiremos nas próximas seções chamarão algumas das chamadas acima em sua implementação do Linux. Sua implementação do Windows pode ser totalmente diferente. Mas nós estaremos olhando para um sistema operacional em geral.

Funções comuns do C

Para encontrar mais informações sobre quaisquer funções, use as páginas man. Observe que as páginas man estão organizadas em seções.Seção 2 são chamadas do sistema. Seção 3 são bibliotecas C. Na web, o Google man 7 open. No shell man -S2 openouman -S3 printf

Entrada / saída

Nesta seção, abordaremos todas as funções básicas de entrada e saída na biblioteca padrão com referências a chamadas do sistema. Como um pouco de terminologia, quando o seu programa está rodando em um terminal, sua saída padrão é o seu terminal e a entrada padrão é o seu terminal aguardando entrada. Há outros casos, como você verá mais adiante neste curso de redirecionamento, para apontar outras coisas. Eles são designados pelos descritores de arquivo 0 e 1, respectivamente. 2 é reservado para erro padrão que por convenção de biblioteca não é bufferizado.

fluxos orientados para stdout

A saída padrão ou fluxos orientados para stdout são fluxos cujas únicas opções são gravar no stdout. printfé a função com a qual a maioria das pessoas está familiarizada nessa

categoria. O primeiro parâmetro é uma cadeia de formato que inclui espaços reservados para os dados a serem impressos. Especificadores de formato comuns são os seguintes

1. %s trate o argumento como um ponteiro de string ac, continue imprimindo todos os caracteres até que o caractere NULL seja atingido
2. %d imprime o argumento como um inteiro
3. %p imprime o argumento como um endereço de memória.

Por padrão, para desempenho, printfna verdade não grava nada até que seu buffer esteja cheio ou uma nova linha seja impressa. Aqui está um exemplo de impressão das coisas. char * name = ... ; int score = ...; printf ("Hello %s, your result is %d \n " , name , score); printf ("Debug: The string and int are stored at: %p and %p \n " , name , & score); // name already is a char pointer and points to the start of the array. // We need "&" to get the address of the int variable
Na seção anterior, printfchama a chamada do sistema write. printfinclui um buffer interno, portanto, para aumentar o desempenho, você printfpode não ligar writetoda vez que ligar printf. Outra peculiaridade do printf é forçada a chamar writeum personagem de nova linha. Se alguém quiser printfchamar escrever sem uma nova linha fflush(FILE* inp). Por último, printfé uma função da biblioteca C. writeé uma chamada de sistema e, como sabemos, as chamadas do sistema são caras. Por outro lado, printfusa um buffer que atende melhor às nossas necessidades naquele momento. Ou seja, não otimize demais o uso do printf muito, mesmo se as chamadas do sistema forem caras - os programas fazem isso o tempo todo.
Para imprimir seqüências de caracteres e caracteres individuais usar puts(name)e putchar(c)onde nome é um ponteiro para uma seqüência de caracteres C e c é apenas umchar
puts ("Current selection: "); putchar ('1');

Outros fluxos

Para imprimir em outros fluxos de arquivos use fprintf(_file_ , "Hello %s, score: %d", name, score);Where _file_ é predefinido 'stdout' 'stderr' ou um ponteiro FILE que foi retornado por fopenou fdopen. Você também pode usar descritores de arquivos na família de funções printf! Apenas usedprintf(int fd, char* format_string, ...); . Basta lembrar que o fluxo pode ser armazenado em buffer por meio do buffer interno, portanto, você precisará assegurar que os dados sejam gravados no descritor de arquivo.
Para imprimir dados em uma string C, use sprintfou melhor snprintf. snprintfretorna o número de caracteres escritos excluindo o byte de encerramento. A seguir, isso seria um máximo de 199. Nós usaríamos sprintfem casos em que sabemos que o tamanho da string não será mais do que uma certa quantia fixa - pense em imprimir um inteiro, ele nunca será mais do que 11 caracteres com o byte nulo.
// Fixed char int_string [20]; sprintf (int_string , "%d" , integer); // Variable length char result [200]; int len = snprintf (result , sizeof (result), "%s:%d" , name , score);

funções orientadas por stdin

Entradas padrão ou funções orientadas por stdin são lidas diretamente do stdin. A maioria dessas funções foram descontinuadas devido a elas serem mal projetadas, por isso tratamos stdin como um arquivo do qual podemos ler bytes. Um dos infratores mais notórios é gets. getsfoi descontinuado no padrão C99 e foi removido do padrão C mais recente

(C11). A razão pela qual ele foi preterido foi que não há como controlar o comprimento, portanto, as strings podem ser invadidas com muita facilidade.
Programas devem usar fgetsou em getlinevez disso. Aqui está um exemplo rápido de ler no máximo 10 caracteres de stdin.
char * fgets (char * str , int num , FILE * stream); ssize_t getline (char ** lineptr , size_t * n , FILE * stream); // Example, the following will not read more than 9 chars char buffer [10]; char * result = fgets (buffer , sizeof (buffer), stdin);
O resultado é NULL se houver um erro ou o final do arquivo for atingido. Note, ao contrário gets, fgetscopia a nova linha no buffer, que você pode querer descartar. Por outro lado, uma das vantagens getlineé a de alocar e realocar automaticamente um buffer no heap de tamanho suficiente.
// ssize_t getline(char **lineptr, size_t *n, FILE *stream); /* set buffer and size to 0; they will be changed by getline */ char * buffer = NULL ; size_t size = 0 ; ssize_t chars = getline (& buffer , & size , stdin); // Discard newline character if it is present, if (chars > 0 && buffer [chars - 1] == '\n') buffer [chars - 1] = '\0' ; // Read another line. // The existing buffer will be re-used, or, if necessary, // It will be `free`'d and a new larger buffer will `malloc`'d chars = getline (& buffer , & size , stdin); // Later... don't forget to free the buffer! free (buffer);
Além dessas funções, temos perrorum significado duplo. Digamos que você tenha uma chamada de função que falhou porque você verificou a página do manual e é um código de retorno com falha. perror(const char* message)irá imprimir a versão em inglês do erro para stderr.
int main (){ int ret = open ("IDoNotExist.txt" , O_RDONLY); if (ret < 0){ perror ("Opening IDoNotExist:"); } //... return 0 ; }
Para ter uma entrada de análise de função de biblioteca além de lê-la, use scanf(ou fscanfou sscanf) para obter entrada do fluxo de entrada padrão, um fluxo de arquivo arbitrário ou uma sequência C, respectivamente. Todas essas funções retornarão quantos itens foram analisados; É uma boa idéia verificar se o número é igual ao valor esperado.
Também naturalmente printf, scanffunções requerem ponteiros válidos. Em vez de apenas apontar para a memória válida, eles também precisam ser graváveis. É uma fonte comum de erro para passar um valor de ponteiro incorreto. Por exemplo,
int * data = malloc (sizeof (int)); char * line = "v 10" ; char type ; // Good practice: Check scanf parsed the line and read two values: int ok = 2 == sscanf (line , "%c %d" , & type , & data); // pointer error
Queríamos escrever o valor do caractere em c e o valor inteiro na memória malloc. No entanto, passamos o endereço do ponteiro de dados, não o que o ponteiro está apontando! Então sscanfvai mudar o ponteiro em si. O ponteiro agora apontará para o endereço 10, de modo que esse código falhará mais tarde quando for liberado (dados).
Agora, o scanf continuará lendo os caracteres até que a string termine. Para impedir que o scanf cause um estouro de buffer, use um especificador de formato. Certifique-se de passar um a menos que o tamanho do buffer.
char buffer [10]; scanf ("%9s" , buffer); // reads up to 9 characters from input (leave room for the 10th byte to be the terminating byte)
Uma última coisa a notar é se você achava que as chamadas do sistema eram caras, a scanffamília é muito mais cara devido a razões de compatibilidade. Imagine precisar pegar todos os especificadores do printf corretamente, o código não será muito eficiente. Na

maioria das vezes, se você estiver escrevendo um programa autônomo, escreva o analisador sozinho. Se for um dos programas, fique à vontade para usar o scanf.

string.h

As funções String.h são uma série de funções que tratam de como manipular e checar pedaços de memória. A maioria deles lida com C-Strings Um C-String é uma série de bytes delimitados por um caractere NUL que é igual ao byte 0x00. https://linux.die.net/man/3/string . Qualquer comportamento que não esteja nos documentos, como passar, strlen(NULL)é considerado um comportamento indefinido.

- int strlen(const char *s) retorna o comprimento da string não incluindo o byte nulo
- int strcmp(const char *s1, const char *s2)retorna um inteiro determinando a ordem lexicográfica das strings. Se s1 onde vir antes de s2 em um dicionário, então -1 é retornado. Se as duas cadeias são iguais, então 0. Else, 1.
- char *strcpy(char *dest, const char *src)Copia a string em srcpara dest. **assume dest tem espaço suficiente para src**
- char *strcat(char *dest, const char *src)Concatena a string srcaté o final do destino. **Esta função assume que há espaço suficiente para srcno final do destino, incluindo o byte NULL**
- char *strdup(const char *dest)Retorna uma malloccópia da string.
- char *strchr(const char *haystack, int needle)Retorna um ponteiro para a primeira ocorrência de needleno haystack. Se nenhum for encontrado, NULLserá retornado.
- char *strstr(const char *haystack, const char *needle) O mesmo que acima, mas desta vez uma string!
- char *strtok(const char *str, const char *delims)

  Uma função perigosa mas útil strtok pega uma string e a simboliza. Isso significa que ele transformará as strings em strings separadas. Esta função tem muitas especificações, então por favor leia as páginas de manual abaixo.
  #include <stdio.h> #include <string.h> int main (){ char * upped = strdup ("strtok,is,tricky,!!"); char * start = strtok (upped , ","); do { printf ("%s \n " , start); } while ((start = strtok (NULL , ","))); return 0 ; }
  Saída
  strtok is tricky !!
  Por que isso é complicado? Bem, o que acontece quando eu mudo uppedassim?
  char * upped = strdup ("strtok,is,tricky,,,!!");
- Para uso de análise de número inteiro long int strtol(const char *nptr, char **endptr, int base);oulong long int strtoll(const char *nptr, char **endptr, int base); .
  O que essas funções fazem é levar o ponteiro para sua string *nptre um ponteiro base(isto é, binário, octal, decimal, hexadecimal, etc) e um ponteiro opcional endptre retorna um valor analisado.
  int main (){ const char * nptr = "1A2436" ; char * endptr ; long int result = strtol (nptr , & endptr , 16); return 0 ; }
  Tenha cuidado embora! O tratamento de erros é complicado porque a função não retornará um código de erro. Se você der a ela uma string que não seja um número, ela retornará 0. Isso significa que você não pode diferenciar entre um "0" válido e

uma string inválida. Veja a man page para mais detalhes sobre o comportamento do strol com valores inválidos e fora dos limites. Uma alternativa mais segura é usar para sscanf(e verificar o valor de retorno).
int main (){ const char * input = "0" ; // or "!##@" or "" char * endptr ; long int parsed = strtol (input , & endptr , 10); if (parsed == 0){ // Either the input string was not a valid base-10 number or it really was zero! } return 0 ; }

- void *memcpy(void *dest, const void *src, size_t n)move nbytes começando em srcpara dest. **Tenha cuidado** , há um comportamento indefinido quando as regiões de memória se sobrepõem. Este é um dos exemplos clássicos de exemplos da minha máquina, porque muitas vezes o valgrind não conseguirá pegá-lo porque parecerá que funciona na sua máquina. Quando o autograder acertar, falhe. Considere a versão mais segura abaixo.
- void *memmove(void *dest, const void *src, size_t n)faz o mesmo que acima, mas se as regiões de memória se sobrepuserem, então é garantido que todos os bytes serão copiados corretamente. memcpye memmoveambos em string.h? Porque seqüências de caracteres são essencialmente memória bruta com um byte nulo no final deles!

Modelo de Memória C

O modelo de memória C é provavelmente diferente da maioria dos que você já viu antes. Em vez de alocar um objeto com segurança de tipo, usamos uma variável automática ou solicitamos uma sequência de bytes com mallocou outro membro da família e, posteriormente, nós free.

Estruturas

Em termos de baixo nível, uma struct é apenas uma peça de memória contígua, nada mais. Assim como uma matriz, uma estrutura tem espaço suficiente para manter todos os seus membros. Mas ao contrário de um array, ele pode armazenar tipos diferentes. Considere a estrutura de contato declarada acima
struct contact { char firstname [20]; char lastname [20]; unsigned int phone ; }; struct contact bhuvan ;
/* a lot of times we will do the following typedef so we can just write contact contact1 */ typedef struct contact contact ; contact bhuvan ; /* You can also declare the struct like this to get it done in one statement */ typedef struct optional_name { ... } contact ;
Se você compilar o código sem quaisquer otimizações e reordenações, poderá esperar que os endereços de cada uma das variáveis tenham essa aparência.
& bhuvan // 0x100 & bhuvan . firstname // 0x100 = 0x100+0x00 & bhuvan . lastname // 0x114 = 0x100+0x14 & bhuvan . phone // 0x128 = 0x100+0x28
Porque tudo o que o seu compilador faz é dizer 'reserve tanto espaço, e eu irei calcular os deslocamentos de quaisquer variáveis que você queira escrever'. Os deslocamentos são onde a variável começa em. As variáveis de telefone começam nos 0x128bytes th e continuam por bytes sizeof (int), mas não sempre. **Offsets não determinam onde a variável termina** . Considere o seguinte hack que você vê em um monte de código do kernel.

typedef struct { int length ; char c_str [0]; } string ; const char * to_convert = "bhuvan" ; int length = strlen (to_convert); // Let's convert to ac string string * bhuvan_name ; bhuvan_name = malloc (sizeof (string) + length + 1); /* Currently, our memory looks like this with junk in those black spaces ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ bhuvan_name = |___|___|___|___|___|___|___|___|___|___|___| */ bhuvan_name -> length = length ; /* This writes the following values to the first four bytes The rest is still garbage ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ bhuvan_name = | 0 | 0 | 0 | 6 |___|___|___|___|___|___|___|___|___| */ strcpy (bhuvan_name -> c_str , to_convert); /* Now our string is filled in correctly at the end of the struct ___ ___ ___ ___ ___ ___ ___ ___ ___ ____ bhuvan_name = | 0 | 0 | 0 | 6 | b | h | u | v | a | n | \0 | ¯ */ strcmp (bhuvan_name -> c_str , "bhuvan") == 0 //The strings are equal!

O que essa matriz de comprimento zero faz é apontar para o **final da estrutura.** Isso significa que o compilador deixará espaço para todos os elementos calculados com relação ao seu tamanho no sistema operacional (ints, chars, etc). A matriz de comprimento zero não ocupará bytes de espaço. Como estruturas são apenas partes contínuas da memória, podemos alocar **mais** espaço do que o necessário e usar o espaço extra como um local para armazenar bytes extras. Embora isso pareça um truque de sala de estar, é uma otimização importante porque para ter uma cadeia de comprimento variável de outra maneira, seria necessário ter duas chamadas de alocação de memória diferentes. Isso é altamente ineficiente para fazer algo tão comum na programação quanto manipulação de strings.

Extra: Embalagem Estrutural

Estruturas podem exigir algo chamado http://www.catb.org/esr/structure-packing/ (tutorial). **Nós não esperamos que você faça o empacotamento de estruturas neste curso, apenas saiba que está lá.** Isto é porque nos primeiros dias (e até agora) quando você tem um endereço de memória você tem que fazê-lo em blocos de 32 ou 64 bits. Isso também significava que você só poderia solicitar endereços que fossem múltiplos disso. Significa que

struct picture { int height ; pixel ** data ; int width ; char * encoding ; }

Você acha que a imagem se parece com isso. Uma caixa tem quatro bytes

| h | data | w | encod | |---+-------+---+-------| |___|___|___|___|___|___|___|

Com o empacotamento struct, conceitualmente ficaria assim

struct picture { int height ; char slop1 [4]; pixel ** data ; int width ; char slop2 [4]; char * encoding ; }

Aqui está um pouco mais de arte ASCII para você

| h | ? | data | w | ? | encod | |---+---+-------+---+---+-------| |___|___|___ ___|___|___|___ ___|

Isso está em um sistema de 64 bits. Isso nem sempre é o caso, porque às vezes seu processador suporta acessos desalinhados. O que isto significa? Bem, existem duas opções que você pode definir um atributo

struct __attribute__ ((packed , aligned (4))) picture { int height ; pixel ** data ; int width ; char * encoding ; }

| h | data | w | encod | |---+-------+---+-------| |___|___|___|___|___|___|___|

Mas agora, toda vez que eu quiser acessar dataou encoding, eu tenho que fazer dois acessos à memória. A outra coisa que você pode fazer é reordenar a estrutura, embora isso nem sempre seja possível

struct picture { int height ; int width ; pixel ** data ; char * encoding ; }
| h | w | data | encod | |---+---+---+---+-------| |___|___|___|___|___|___|

Cordas em C

Em C, temos https://en.wikipedia.org/wiki/Null-terminated_string strings em vez de https://en.wikipedia.org/wiki/String_(computer_science)#Length-prefixed por motivos históricos. O que isso significa para a sua programação diária média é que você precisa lembrar o caractere nulo! Uma string em C é definida como um grupo de bytes até que você atinja '' ou o NUL Byte.

Lugares para cordas

Sempre que você define uma string constante - uma na forma char* str = "constant"- essa string é armazenada nos *dados,* dependendo de sua arquitetura, que é **somente leitura,** o que significa que qualquer tentativa de modificar a string causará um SEGFAULT. Pode-se também declarar que as strings estão no segmento de dados graváveis ou na pilha. Para fazer isso, basta especificar um comprimento para a string ou colocar colchetes em vez de um ponteiro char str[] = "mutable"e colocar no escopo global ou no escopo da função para o segmento de dados ou a pilha, respectivamente. Se, no entanto, houver mallocespaço, pode-se mudar essa string para ser o que quiserem. Esquecer a NUL terminar uma string é um grande efeito nas cordas! A verificação dos limites é importante. O bug heartbleed mencionado anteriormente no livro é parcialmente por causa disso.
Strings em C são representados como caracteres na memória. O final da string inclui um byte NUL (0). Então, "ABC" requer quatro (4) bytes. A única maneira de descobrir o comprimento de uma string C é continuar lendo a memória até encontrar o byte NULL. Caracteres C são sempre exatamente um byte cada.

Constantes de string são constantes

Uma constante de string é naturalmente constante. Qualquer gravação fará com que o sistema operacional produza um SEGFAULT.
char array [] = "Hi!" ; // array contains a mutable copy strcpy (array , "OK"); char * ptr = "Can't change me" ; // ptr points to some immutable memory strcpy (ptr , "Will not work");
Literais de cadeia são matrizes de caracteres armazenadas no segmento de código do programa, que é imutável. Dois literais de string podem compartilhar o mesmo espaço na memória. Um exemplo segue.
char * str1 = "Bhuvy likes books" ; char * str2 = "Bhuvy likes books" ;
As cadeias apontadas por str1e str2podem realmente residir no mesmo local na memória. Chararrays, no entanto, contêm o valor literal que foi copiado do segmento de código para a pilha ou memória estática. Estes seguintes matrizes de char não residem no mesmo lugar na memória.
char arr1 [] = "Bhuvy also likes to write" ; char arr2 [] = "Bhuvy also likes to write" ;
Aqui estão algumas maneiras comuns de inicializar uma string include. Onde eles residem na memória?
char * str = "ABC" ; char str [] = "ABC" ; char str [] = { 'A' , 'B' , 'C' , '\0' };
char ary [] = "Hello" ; char * ptr = "Hello" ;

Também podemos imprimir o ponteiro e o conteúdo da cadeia ac com muita facilidade. Aqui está um código clichê para fazer a impressão.
char ary [] = "Hello" ; char * ptr = "Hello" ; // Print out address and contents printf ("%p : %s \n " , ary , ary); printf ("%p : %s \n " , ptr , ptr);
Como mencionado anteriormente, o array char é mutável, então podemos mudar seu conteúdo. Tenha cuidado para não escrever bytes além do final da matriz. Também, novamente, tenha cuidado para não misturar os dois.
strcpy (ary , "World"); // OK strcpy (ptr , "World"); // NOT OK - Segmentation fault (crashes)
Podemos, no entanto, ao contrário do array, mudamos ptrpara apontar para outro pedaço de memória,
ptr = "World" ; // OK! ptr = ary ; // OK! ary = "World" ; // NO won't compile // ary is doomed to always refer to the original array. printf ("%p : %s \n " , ptr , ptr); strcpy (ptr , "World"); // OK because now ptr is pointing to mutable memory (the array)
O que tirar disso é que os ponteiros * podem apontar para qualquer tipo de memória, enquanto os arrays de caracteres mencionados anteriormente sempre se referirão à mesma parte da memória. Em um caso mais comum, os ponteiros apontarão para a memória heap, em cujo caso a memória referida pelo ponteiro **pode** ser modificada.

Ponteiros

Até agora, você pode não ter tido muito trabalho com ponteiros. Ponteiros são variáveis que contêm endereços. Esses endereços têm valor numérico, mas geralmente nos preocupamos com o conteúdo abaixo. Nesta seção, tentaremos levá-lo a uma introdução básica de ponteiros.

Noções básicas de ponteiro

Declarando um ponteiro

Um ponteiro refere-se a um endereço de memória. O tipo do ponteiro é útil - diz ao compilador quantos bytes precisam ser lidos / gravados e quais são as semânticas para adição.
int * ptr1 ; char * ptr2 ;
Devido à gramática de C, um int*ou qualquer ponteiro não é realmente seu próprio tipo. Você precisa preceder cada variável de ponteiro com um asterisco. Como uma pegadinha comum, o seguinte
int * ptr3 , ptr4 ;
Apenas irá declarar *ptr3como um ponteiro. ptr4será realmente uma variável int regular. Para corrigir essa declaração, mantenha a *precedente para o ponteiro
int * ptr3 , * ptr4 ;
Tenha isso em mente para as estruturas também. Se alguém não os digitar, então o ponteiro vai após o tipo.
struct person * ptr3 ;

Leitura / Escrita com ponteiros

Vamos dizer que declaramos um ponteiro int ptr. Por uma questão de discussão, digamos que ptrcontém o endereço de memória 0x1000. Se quisermos escrever para um ponteiro, podemos desreferenciar e atribuir *ptr.
* ptr = 0 ; // Writes some memory.
O que C vai fazer é tomar o tipo de ponteiro que é um inte escrever sizeof(int)bytes desde o início do ponteiro, o que significa que bytes 0x1000, 0x1001, 0x1002, 0x1003tudo será zero. O número de bytes escritos depende do tipo de ponteiro. É o mesmo para todos os tipos primitivos, mas as estruturas são um pouco diferentes.
A leitura funciona praticamente da mesma maneira, exceto que você coloca a variável no ponto que precisa do valor.
int double = * ptr * 2
Ler e escrever para tipos não-primitivos é um pouco complicado. Você não pode atribuir um ponteiro a um inteiro, você precisa ter o mesmo tipo. Além disso, a unidade de compilação - geralmente o arquivo ou cabeçalho - precisa ter o tamanho da estrutura de dados prontamente disponível, o que significa que as estruturas de dados opacas não podem ser copiadas. Aqui está um exemplo de atribuição de um ponteiro de estrutura
#include <stdio.h> typedef struct { int a1 ; int a2 ; } pair ; int main () { pair obj ; pair zeros ; zeros . a1 = 0 ; zeros . a2 = 0 ; pair * ptr = & obj ; obj . a1 = 1 ; obj . a2 = 2 ; * ptr = zeros ; printf ("a1: %d, a2: %d \n " , ptr -> a1 , ptr -> a2); return 0 ; }
Quanto a leitura de ponteiros de estrutura, não faça isso diretamente.

Aritmética de ponteiro

Além de adicionar a um inteiro, você pode adicionar um inteiro a um ponteiro. No entanto, o tipo de ponteiro é usado para determinar quanto incrementar o ponteiro. Para ponteiros de caracteres, isso é trivial porque os caracteres são sempre um byte.
char * ptr = "Hello" ; // ptr holds the memory location of 'H' ptr += 2 ; //ptr now points to the first 'l'
Se um int é de 4 bytes, então ptr + 1 aponta para 4 bytes após o que ptr está apontando.
char * ptr = "ABCDEFGH" ; int * bna = (int *) ptr ; bna += 1 ; // Would cause iterate by one integer space (ie 4 bytes on some systems) ptr = (char *) bna ; printf ("%s" , ptr); /* Notice how only 'EFGH' is printed. Why is that? Well as mentioned above, when performing 'bna+=1' we are increasing the **integer** pointer by 1, (translates to 4 bytes on most systems) which is equivalent to 4 characters (each character is only 1 byte)*/ return 0 ;
Como a aritmética de ponteiros em C é sempre automaticamente dimensionada pelo tamanho do tipo que é apontado, você não pode executar a aritmética de ponteiros em ponteiros nulos pelos padrões POSIX, embora os compiladores tratem o tipo subjacente como char. Você pode pensar em aritmética de ponteiro em C como essencialmente fazendo o seguinte
int * ptr1 = ...; int * offset = ptr1 + 4 ;
Pensar
int * ptr1 = ...; char * temp_ptr1 = (char *) ptr1 ; int * offset = (int *)(temp_ptr1 + sizeof (int) * 4);
Para obter o valor. **Toda vez que você fizer aritmética com ponteiro, respire fundo e certifique-se de estar mudando o número de bytes que pensa estar mudando.**

Então, o que é um ponteiro vazio?

Um ponteiro vazio é um ponteiro sem um tipo. Os ponteiros vazios são usados quando um tipo de dados com o qual você está lidando é desconhecido ou quando você faz interface com o código C com outras linguagens de programação. Você pode pensar nisso como um ponteiro bruto ou apenas um endereço de memória. Você não pode ler ou gravar diretamente nele porque o tipo void não tem tamanho. malloc por padrão retorna um ponteiro vazio que pode ser promovido com segurança para qualquer outro tipo.
void * give_me_space = malloc (10); char * string = give_me_space ;
Isso não requer uma conversão porque C promove automaticamente void*para seu tipo apropriado. **Nota:**
gcce clangnão são totalmente compatíveis com ISO-C, o que significa que eles permitirão que você faça aritmética em um ponteiro vazio. Eles vão tratá-lo como um char *ponteiro. Não faça isso porque pode não funcionar com todos os compiladores!

Erros comuns

Bytes nulos

O que há de errado com esse código?
void mystrcpy (char * dest , char * src) { // void means no return value while (* src) { dest = src ; src ++ ; dest ++ ; } }
No código acima, simplesmente altera o ponteiro dest para apontar para a string de origem. Além disso, os bytes NULs não são copiados. Aqui está uma versão melhor -
while (* src) { * dest = * src ; src ++ ; dest ++ ; } * dest = * src ;
Note que também é comum ver o seguinte tipo de implementação, que faz tudo dentro do teste de expressão, incluindo a cópia do byte NUL.
while ((* dest ++ = * src ++)) {};

Dupla Liberta

Um erro duplo livre é quando você acidentalmente tenta liberar a mesma alocação duas vezes.
int * p = malloc (sizeof (int)); free (p); * p = 123 ; // Oops! - Dangling pointer! Writing to memory we don't own anymore free (p); // Oops! - Double free!
A correção é a primeira a escrever programas corretos! Em segundo lugar, é uma boa higienização de programação para redefinir os ponteiros uma vez que a memória tenha sido liberada. Isso garante que o ponteiro não possa ser usado incorretamente sem que o programa falhe.
p = NULL ; // Now you can't use this pointer by mistake

Retornando ponteiros para variáveis automáticas

int * f () { int result = 42 ; static int imok ; return & imok ; // OK - static variables are not on the stack return & result ; // Not OK }
Variáveis automáticas são obrigadas a empilhar a memória apenas durante a vida útil da função. Depois que a função retorna, é um erro continuar usando a memória.

Alocação de memória insuficiente

struct User { char name [100]; }; typedef struct User user_t ; user_t * user = (user_t *) malloc (sizeof (user));
No exemplo acima, precisamos alocar bytes suficientes para a estrutura. Em vez disso, alocamos bytes suficientes para manter um ponteiro. Quando começarmos a usar o ponteiro do usuário, corromperemos a memória. O código correto é mostrado abaixo.
struct User { char name [100]; }; typedef struct User user_t ; user_t * user = (user_t *) malloc (sizeof (user_t));

Estouro / estouro de buffer

Exemplo famoso: Heart Bleed executou um memcpy em um buffer que era de tamanho insuficiente. Exemplo simples: implemente um strcpy e esqueça de adicionar um ao strlen, ao determinar o tamanho da memória requerida.
#define N (10) int i = N , array [N]; for (; i >= 0 ; i --) array [i] = i ;
C não verifica se os ponteiros são válidos. O exemplo acima escreve no array[10]qual está fora dos limites da matriz. Isso pode causar corrupção de memória porque esse local de memória provavelmente está sendo usado para outra coisa. Na prática, isso pode ser mais difícil de detectar porque o estouro / estouro negativo pode ocorrer em uma chamada da biblioteca. Aqui está o nosso velho amigo fica.
gets (array); // Let's hope the input is shorter than my array!

Strings requerem strlen (s) +1 bytes

Cada string deve ter um byte nulo após os últimos caracteres. Para armazenar a string “Hi”, são necessários 3 bytes: [H] [i] [\ 0].
char * strdup (const char * input) { /* return a copy of 'input' */ char * copy ; copy = malloc (sizeof (char *)); /* nope! this allocates space for a pointer, not a string */ copy = malloc (strlen (input)); /* Almost...but what about the null terminator? */ copy = malloc (strlen (input) + 1); /* That's right. */ strcpy (copy , input); /* strcpy will provide the null terminator */ return copy ; }

Usando variáveis não inicializadas

int myfunction () { int x ; int y = x + 2 ; ...
Variáveis automáticas mantêm o lixo ou o padrão de bits que estavam na memória ou no registrador. É um erro supor que será sempre inicializado como zero.

Assumindo que a memória não inicializada será zerada

void myfunct () { char array [10]; char * p = malloc (10);
Automático (variáveis temporárias) não são automaticamente inicializados para zero. As alocações de heap usando malloc não são inicializadas automaticamente para zero.

Erros de fluxo lógico e programa

Estes são um conjunto de erros que podem ou não fazer sentido dentro do contexto do programa.

Igualdade vs. Igualdade

Confusamente em C, o operador de atribuição também retorna o valor atribuído. Na maioria das vezes, é ignorado. Podemos usá-lo para inicializar várias coisas na mesma linha.
int p1 , p2 ; p1 = p2 = 0 ;
Mais confusamente, se esquecermos um sinal de igual no operador de igualdade, acabaremos atribuindo essa variável. Na maioria das vezes isso não é o que queremos fazer.
int answer = 3 ; // Will print out the answer. if (answer = 42) { printf ("I've solved the answer! It's %d" , answer);}
A maneira rápida de consertar isso é adquirir o hábito de colocar as constantes primeiro.
if (42 = answer) { printf ("I've solved the answer! It's %d" , answer);}
Há casos em que queremos fazer isso. Um exemplo comum é o getline.
while ((nread = getline (& line , & len , stream)) != - 1)
Essa parte do código chama getline e atribui o valor de retorno ou o número de bytes lidos para nread. Ele também na mesma linha verifica se esse valor é -1 e, se assim for, termina o loop. É sempre uma boa prática colocar parênteses em torno de qualquer condição de atribuição.

Funções não declaradas ou incorrectamente prototipadas

Você pode ver o código fazer algo assim.
time_t start = time ();
A função do sistema 'tempo' realmente leva um parâmetro a um ponteiro para alguma memória que pode receber a estrutura time_t ou NULL. O compilador não detectou esse erro porque o programador não forneceu um protótipo de função válido incluindo time.h. Mais confusamente, isso poderia compilar, funcionar por décadas e depois travar. A razão para isso é que o tempo seria encontrado no momento do link, não no tempo de compilação na biblioteca padrão c que quase certamente já está na memória. Como não estamos passando um parâmetro, esperamos que os argumentos na pilha (qualquer lixo) sejam zerados porque, se não for, o tempo tentará gravar o resultado da função nesse lixo, o que causará o programa. para SEGFAULT.

Ponto-e-vírgula extra

Este é um bem simples, não coloque ponto e vírgula quando não for necessário.
for (int i = 0 ; i < 5 ; i ++) ; printf ("I'm printed once"); while (x < 10); x ++ ; // X is never incremented
No entanto, o código a seguir está perfeitamente correto.
for (int i = 0 ; i < 5 ; i ++){ printf ("%d \n " , i);;;;;;;;;;;;; }
Não há problema em ter esse tipo de código, porque a linguagem C usa ponto e vírgula (;) para separar instruções. Se não houver nenhuma declaração entre ponto-e-vírgula, então não há nada para fazer e o compilador avança para a próxima instrução. Para economizar muita confusão, mantenha-se sempre usando chaves. Aumenta o número de linhas de código, o que é uma excelente métrica de produtividade.

Tópicos

- Representação C Strings
- C Strings como ponteiros
- char p [ ] vs char * p
- Funções simples de string C (strcmp, strcat, strcpy)
- tamanho de char
- sizeof x vs x *
- Vida útil da memória heap
- Chamadas para alocação de heap
- Ponteiros de referência
- Operador de endereços
- Aritmética ponteiro
- Duplicação de cordas
- Truncagem de cadeia
- erro duplo-livre
- Literais string
- Imprimir formatação.
- erros de memória fora dos limites
- memória estática
- saída de entrada do arquivo. Biblioteca POSIX vs. C
- Saída de entrada C: fprintf e printf
- Arquivo POSIX IO (ler, escrever, abrir)
- Buffering de stdout

Perguntas / Exercícios

- O que o seguinte imprime?
  int main (){ fprintf (stderr , "Hello "); fprintf (stdout , "It's a small "); fprintf (stderr , "World \n "); fprintf (stdout , "place \n "); return 0 ; }
- Quais são as diferenças entre as duas declarações a seguir? O que sizeofretorna para um deles?
  char str1 [] = "bhuvan" ; char * str2 = "another one" ;
- O que é uma string em c?
- Codifique um simples my_strcmp. Que tal my_strcat, my_strcpyou my_strdup? Bônus: codifique as funções enquanto percorre as strings apenas *uma vez* .
- O que deve o seguinte geralmente retornar?
  int * ptr ; sizeof (ptr); sizeof (* ptr);
- O que é malloc? Como isso é diferente de calloc. Uma vez que a memória é malloced como posso usar realloc?
- Qual é o &operador? Que tal *?
- Aritmética de Ponteiro. Assuma os seguintes endereços. Quais são os seguintes turnos?

char ** ptr = malloc (10); //0x100 ptr [0] = malloc (20); //0x200 ptr [1] = malloc (20); //0x300
    - ptr + 2
    - ptr + 4
    - ptr[0] + 4
    - ptr[1] + 2000
    - *((int)(ptr + 1)) + 3

- Como evitar erros duplos gratuitos?
- Qual é o especificador printf para imprimir uma string int, ou char?
- O código a seguir é válido? Se sim, porque?Onde está outputlocalizado?
  char * foo (int var){ static char output [20]; snprintf (output , 20 , "%d" , var); return output ; }
- Escreva uma função que aceite uma string e abra esse arquivo imprime o arquivo 40 bytes por vez, mas todas as outras impressões invertem a string (tente usar a API POSIX para isso).
- Quais são algumas diferenças entre o modelo de descritor de arquivo POSIX e os C's FILE*(ou seja, quais chamadas de função são usadas e quais são armazenadas em buffer)? O POSIX usa C's FILE*internamente ou vice-versa?

Rapid Fire: Pointer Arithmetic

A aritmética de ponteiro é realmente importante! Você precisa saber quantos bytes cada ponteiro é movido com cada adição. O seguinte é uma seção de fogo rápido. Nós vamos usar as seguintes definições
int * int_ ; // sizeof(int) == 4; long * long_ ; // sizeof(long) == 8; char * char_ ; int * short_ ; //sizeof(short) == 2; int ** int_ptr ; // sizeof(int*) == 8;
Quantos bytes são movidos das seguintes adições?

1. int_ + 1
2. long_ + 7
3. short_ - 6
4. short_ - sizeof(long)
5. long_ - sizeof(long) + sizeof(int_)
6. long_ - sizeof(long) / sizeof(int)

7. (char*)(int_ptr + sizeof(long)) + sizeof(int_)

Soluções de Incêndio Rápido

1. 4
2. 56
3. -12
4. -16
5. 0
6. -16
7. 72

Duff, Tom. nd “Tom Duff no dispositivo de Duff”. * Tom Duff no dispositivo de Duff *. https://www.lysator.liu.se/c/duffs-device.html .
“Norma IEEE para Aritmética de Ponto Flutuante.” 2008. * Norma IEEE 754-2008 *, 1 a 70 de agosto. https://doi.org/10.1109/IEEESTD.2008.4610935 .
Inc., Apple. 2017. “XNU Kernel.” * Repositório do GitHub *. https://github.com/apple/darwin-xnu ; GitHub.
“Norma ISO C”. 2005. Padrão. Genebra, CH: Organização Internacional para Padronização. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf .
Kernighan, BW e DM Ritchie. 1988. * A linguagem de programação c *. Série de software de computador Prentice-Hall. Prentice Hall. https://books.google.com/books?id=161QAAAAMAAJ .
Amor, Robert. 2010. * Desenvolvimento do Kernel do Linux *. 3 ed. Profissional da Addison-Wesley.
Ritchie, Dennis M. 1993. “O Desenvolvimento da Linguagem c.” * SIGPLAN Not. * 28 (3): 201–8. https://doi.org/10.1145/155360.155580 .