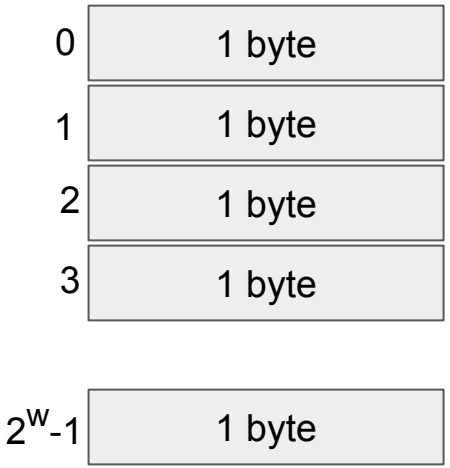
Ponteiros

Programação Estruturada - Cris Sato

Memória

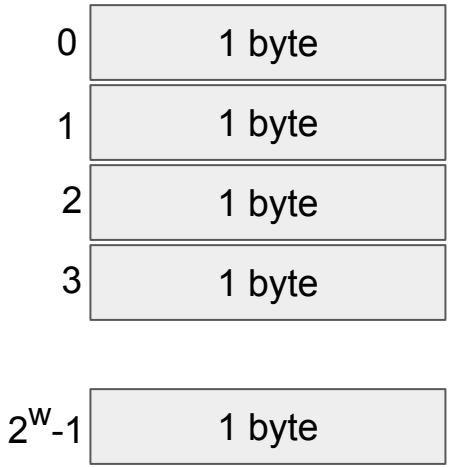
- Vamos assumir que o *word size* é 32 bits. Defina w := 32
- O seu programa tem acesso a uma memória virtual que está organizada em 2^w bloquinhos de 1 byte (8 bits).



- Você pode pensar na memória como um vetor enorme como ao lado
- Chamamos os índices desse vetor de **endereços da memória**

Variáveis na memória

- Um `char` tem 1 byte. Então uma variável `char` ocupa 1 posição na memória.
- Um `int` tem 4 bytes. Então uma variável `int` ocupa 4 posições **consecutivas** na memória.
- A linguagem C nos permite manipular a memória muito mais livremente que algumas outras linguagens
- Para acessar o endereço de uma variável x, basta escrever &x
- Toda variável que criamos tem um endereço



Exemplo: `scanf("%c", &x)`

- Por que mandamos o endereço de x para o scanf ao invés de x?
- Vimos que quando passamos x como parâmetro o que é passado para a função é uma **cópia** de x.
- Mas o scanf tem que mudar o valor de x. Como ele poderia fazer isso se ele só tem uma cópia de x?
- Então passamos para o scanf o endereço de x!
- Assim ele sabe que, se ele mudar o valor armazenado nesse endereço, o valor de x será mudado!
- O interessante é que passamos uma **cópia** do endereço de x para o scanf.
- A passagem de parâmetros em C é **sempre por valor** (ou seja, via cópia).

Exemplo: imprimindo endereços

- Para imprimir um endereço, usamos %p. O endereço será impresso como um número em hexadecimal

```
char x = 'a';
printf("Endereço de x: %p\n", &x);
printf("Valor de x: %c\n", x);
```

Endereço de x:
0x7ffde91adbcf
Valor de x: a

Ponteiros: variáveis que guardam endereços

- `char *y`: é uma variável que guarda um endereço para um `char`
- Para acessar o char que está no endereço `y`, usamos `*y`.
- A variável `y` é chamada de ponteiro pois guarda um endereço. Ou seja, ela **aponta** para um bloquinho da memória

```
char x = 'a';
char *y = &x;
printf("Valor de y: %p\n", y);
printf("Valor guardado no
endereço: %c\n", *y);
```

Valor de y:
0x7ffde91adbcf
Valor guardado
no endereço: a

& e * são os operadores essenciais que manipulam ponteiros!

Então qual é a dificuldade?

- A idéia é realmente bem simples, mas podemos fazer coisas interessantes:
- Abaixo `z` guarda o endereço de `y`

```
char x = 'a';
char *y = &x;
char **z = &y;
printf("Valor de z:
%p\n", z);
printf("Valor de y:
%p\n", y);
```

Valor de z: 0x7ffe3cb226e0
Valor de y: 0x7ffe3cb226df

Passagem de parâmetro

```
int f(int x) {
    x = 2;
}
```

```
-- main --
int x = 3;
printf("x = %d\n", x);
f(x);
printf("x = %d\n", x);
```

```
x = 3
x = 3
```

Não altera o valor de x

Passagem de parâmetro

```
int f(int *x) {
    *x = 2;
}
```

```
-- main --
int x = 3;
printf("x = %d\n", x);
f(&x);
printf("x = %d\n", x);
```

```
x = 3
x = 2
```

Altera o valor de x

Operação de soma e subtração

```
char x = 'a';
char y = 'c';
char *z;
printf("Endereco de x: %p\n", &x);
printf("Endereco de y: %p\n", &y);
z = &x;
printf("%c\n", *z);
z++;
printf("%c\n", *z);
z--;
printf("%c\n", *z);
```

Endereco de x:
0x7ffd8be0a4c6
Endereco de y:
0x7ffd8be0a4c7
a
c
a

Este comportamento não é garantido, poderia ser que x e y não tivessem endereços consecutivos

Operação de soma e subtração

- Ou seja, funciona como em vetores: somar avança posições no vetor e subtrair recua posições.

```
int x = 123, y = 456;
int *z;
printf("Endereco de x: %p\n", &x);
printf("Endereco de y: %p\n", &y);
```

Endereco de x:
0x7fff42b12dd0
Endereco de y:
0x7fff42b12dd4

A diferença aqui é 4

Operação de soma e subtração

```
z = &x;
printf("%d\n", *z);
z++;
printf("%d\n", *z);
z--;
printf("%d\n", *z);
```

```
123
456
123
```

A gente somou 1 mas ele andou 4 posições???
Por que?

Operação de soma e subtração

- No primeiro exemplo, z guardava um endereço para um char, que ocupa apenas 1 bloquinho. Ao fazermos z++, ele anda 1 bloquinho.
- No segundo exemplo, z guardava um endereço para um int, que ocupa 4 bloquinhos. Ao fazermos z++, ele anda 4 bloquinhos.
- Ou seja, quanto andamos depende do tipo de z.
- Isso pode ser **muito útil**!

Vetores

- Quando alocamos um vetor, os bloquinhos são alocados consecutivamente.

```
int v[3];
int *x;
v[0]=123;
v[1]=456;
v[2]=789;
x = v;
printf("*x=%d\n",*x);
x++;
printf("*x=%d\n",*x);
x++;
printf("*x=%d\n",*x);
```

```
*x=123
*x=456
*x=789
```

Vetores

- Quando alocamos um vetor, os bloquinhos são alocados consecutivamente.

```
int v[3];
int *x;
v[0]=123;
v[1]=456;
v[2]=789;
x = v;
printf("x[0]=%d\n",x[0]);
printf("x[1]=%d\n",x[1]);
printf("x[2]=%d\n",x[2]);
```

```
x[0]=123
x[1]=456
x[2]=789
```

x[0] é o mesmo que *x
x[1] é o mesmo que *(x+1)
x[2] é o mesmo que *(x+2)

Vetor como parâmetro

```
void muda(int *x) {
    x[0]=111;
    x[1]=222;
}
```

```
-- main --
int v[2];
v[0]=123;
v[1]=456;
printf("%d %d\n", v[0],v[1]);
muda(v);
printf("%d %d\n", v[0],v[1]);
```

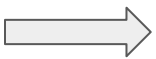
```
123 456
111 222
```

Coisas bizarras

- Como podemos mexer em várias posições da memória pode ser que mudemos uma variável acidentalmente

```
void muda(int *x) {
    x[0]=111;
    x[1]=222;
}
```

```
-- main --
int a = 1;
int b = 2;
printf("%d %d\n", a, b);
muda(&a);
printf("%d %d\n", a, b);
```



```
1 2
111 222
```

O valor de b mudou!

Este comportamento não é garantido, poderia ser que x e y não tivessem endereços consecutivos

Coisas legais

```
int v[3] = {12,34,56};
int *x;
x = v;
x++;
printf("x[-1]=%d\n", x[-1]);
printf("x[0]=%d\n", x[0]);
printf("x[1]=%d\n", x[1]);
```

```
x[-1]=12
x[0]=34
x[1]=56
```

Um vetor com índices negativos!

Alocando um vetor sem saber o seu tamanho

```
int n;
scanf("%d", &n);
char v[n];
```

- É permitido no padrão C99
- Não é permitido no padrão C90. Mas o GCC aceita (é uma extensão).

```
int n;
char *v;
scanf("%d", &n);
v = (char *) malloc(n);
```

- Estamos pedindo para o programa alocar n bloquinhos consecutivos
- Usamos a biblioteca stdlib.h

malloc

- (char *) malloc(n);
- Por que precisamos do (char *)? Na verdade, só precisamos disso para o C++ (não para o C)
- O malloc devolve um ponteiro do tipo **void *,** pois ele não sabe qual tipo de ponteiro estamos pedindo.
- O (char *) é o que chamamos de *typecasting*: estamos convertendo um tipo de variável para outro.
- O parâmetro é o número de bloquinhos que queremos alocar

malloc

- Mas um int precisa de 4 bloquinhos (ou mais, pois o tamanho do int depende do compilador)
- sizeof(tipo) devolve quantos bytes o tipo usa

```
int n;
int *x;
scanf("%d", &n);
x = (int *) malloc(sizeof(int)*n);
```

free(x)

- libera o espaço alocado para x

```
int n;
int *x;
scanf("%d", &n);
x = (int *) malloc(sizeof(int)*n);
for(i=0; i<n; i++) x[i]=i*i;
for(i=0; i<n; i++) printf("%d\n", x[i]);
free(x);
```