Por Vitor Maia
@vitorcmaia
www.dcc.ufrj.br/~vitormaia

Programação C++ para Jogos

Índice – Apresentação I

Introdução 3
Revisão de C 5
Novidades de C++ 7
Struct em C 11
Classes 17
Construtores 20
Encapsulamento 25
Sobrecarga de Métodos 30
Arquivos .h e .cpp 31
Ponteiros em C e C++ 35
Destrutores 38
Referência 40

- É suposto que o leitor já saiba programar (mesmo que mal) em alguma linguagem. Preferencialmente em C.

- Este material não ensinará a fazer jogos, apenas dará mais ênfase às ferramentas presentes na linguagem C++ que são importantes, ou úteis, na programação de um jogo.

- A parte da apresentação sobre Orientação a Objetos será completa. Se o seu foco é aprender OO, este material pode ser usado como referência.

-Apesar do nome do material, este pode ser usado como fonte de estudo mesmo que seu interesse não seja a programação de jogos.

- Quando os includes forem omitidos nos exemplos, suponha que as únicas linhas que faltam são "#include <iostream>" e "using namespace std;"

- Será usado como compilador o **g++**. Se você estiver no Linux, é muito provável que o **g++** já esteja presente. Caso contrário, se vire. Você está num Linux e certamente sabe se virar.

- Caso você use Windows, baixe o Minimalist Gnu for Windows (MinGW), que vem com o **g++**, deste link: http://migre.me/5iPqm

- Não esqueça de configurar o Path, nas variáveis de ambiente do Windows, com a pasta bin do MinGW. Aqui explica como: http://migre.me/5iPyS

- Este comando é usado para compilar o código **C++** com o **g++**. É a mesma linha tanto no Linux quanto no Windows: **g++ -o programa programa.cpp -Wall**

- Para executar o programa no Windows, apenas escreva o nome do programa no Prompt. No Linux, adicione ./ antes. Exemplo: ./programa

- **.cpp** é a extensão de códigos em C++.

O que acha de começarmos com uma revisão de C?
Mas revisão REVISÃO mesmo.

- Para perguntar uma informação ao usuário, usa-se a função scanf.
- Para escrever um número e/ou frase na tela, usa-se a função printf.

- Para que o programa decida entre duas ou mais opções, de acordo com condições, existem os comandos if, switch e ternário.
- Para que o programa repita uma mesma ação várias vezes, existem os comandos for, while e do while.

- Uma variável pode ser um número inteiro (int), um número real (float) ou uma letra (char).

- Se você não sabe do que eu estou falando, feche esta apresentação e compre este livro: http://migre.me/5iPS0

Exemplo de código em C. Este programa escreve na tela os números pares entre 1 e 20.

```
#include <stdio.h>

int main(void)
{
    int i;
    for(i = 1 ; i <= 20 ; i++)
    {
        // '%' é o resto da divisão.
        if(i % 2 == 0)
        {
            printf("%d ",i);
        }
    }
    return 0;
}
```

Considerando-se apenas os programas estruturados (sem classes, com início, meio e fim, como o programa anterior), o que C++ traz de novo em relação a C?

1) É possível criar “int i” diretamente na declaração do for:

```
for(int i = 1 ; i <= 20 ; i++) { ... }
```

Vantagem: Ao fim do bloco do for, esta variável é desalocada.
Em conjunto com o modificador register, que coloca a variável num registrador, o for torna-se muito mais rápido, e sua memória nem fica sabendo que existe um programa sendo executado:

```
for(register int i = 1 ; i <= 20 ; i++) { ... }
```

2) Em C, quando queremos dizer que uma variável é verdadeira ou falsa, precisamos usar um int, e dar os valores 0 para falso e 1 para verdadeiro. O problema disso é que o tipo int ocupa 16 bits, pode ir aproximadamente de -32000 até 32000. Criamos um número desta magnitude para usar apenas os valores 0 e 1...

Em C++ existe o tipo bool, que pode valer apenas true ou false.

```
bool b = true;
if(b)
{
    ...
}
```

3) Os comandos printf e scanf são complicados, e o programador precisa avisar a cada invocação quais tipos está passando por parâmetro. %d para int, %f para float, %c para char, %ld e %lf para long int e double, %s para vetor de char, fora muitos outros.
Mas em C++ existem os comandos cin e cout, para entrada e saída, que dispensam estes símbolos. \o/

```
#include <iostream>
using namespace std;
// As linhas acima são necessárias.
int main(void)
{
    int i;
    cout << "Digite um numero: ";
    cin >> x;
    cout << "Seguinte: " << x + 1 << endl;
    // endl é o mesmo que \n
    return 0;
}
```

4) Para criar uma frase em C, é necessário fazer um vetor ou ponteiro de char. As funções que envolvem manipulação de ponteiro de char possuem parâmetros complicados... mas em C++ existe um tipo novo para manipulação de cadeias de caracteres: string!

```
#include <iostream>
using namespace std;

int main(void) {
// Você sabe como descobrir o tamanho de um char* ?
// Nem eu. Mas para descobrir o tamanho de uma
// string, é assim:
    string s = "Oi eu sou Goku";
    cout << s.size() << endl;

    return 0;
}
```

"Mas como assim? Estou chamando uma função direto de uma variável?"

Sim, e em breve você aprenderá a fazer o mesmo nos seus programas.

Fim da brincadeira... Agora vamos aprender Orientação a Objetos. Mas antes disso, proponho um exercício.

Faça um programa, em C, que jogue dois dados e some os números que saírem.

A minha solução está no próximo slide.

Solução:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    short int dado1, dado2, soma;
    srand (time(NULL));

    dado1 = (short int) (rand() % 6) + 1;
    dado2 = (short int) (rand() % 6) + 1;
    soma = dado1 + dado2;

    printf("%d\n", dado1);
    printf("%d\n", dado2);
    printf("%d\n", soma);

    return 0;
}
```

A solução anterior resolve o problema, mas não é elegante. Ao dar este código para outra pessoa, ela terá que analisar linha a linha para entender o que está acontecendo.

Aponto alguns problemas:

1) Se eu estou simulando um dado, porque eu crio um `int`, em vez de um `Dado`?

2) Sempre que eu quiser sortear um número, eu devo repetir a linha `(rand() % 6) + 1`. Não seria mais fácil simplesmente mandar sortear e pronto, com uma função?

3) Alguém sabe usar o srand sem procurar no Google? Não seria mais interessante se ele já viesse embutido ao chamar a função 'sortear'?

Antes de mostrar a solução em C++, é importante relembrar uma ferramenta de C...

Struct!

Existe em C uma ferramenta chamada Struct. Com esta ferramentas é possível criar um tipo novo, que na prática é um conjunto de variáveis.

Por exemplo, esta é uma Struct que cria uma Pessoa no seu programa em C:

```
typedef struct {
    char* nome;
    int idade;
    float peso;
} Pessoa;

int main(void) {
    Pessoa p;
    p.nome = "Maria";
    p.idade = 30;
    p.peso = 75.5;
    printf("%s %d %g",p.nome,p.idade,p.peso);

    return 0;
}
```

Com Structs é possível definir características de uma pessoa (nome, idade, peso), mas não é possível definir **ações** que esta pessoa pode praticar (andar, falar).

Mas **e se** fosse possível? Vamos **fingir** que Structs permitem a definição de ações. Entenda 'ações' como simples funções de C.

Caso fosse possível, ficaria assim:

```
typedef struct {
    char* nome;
    void falar(void) {
        printf("Oi, meu nome eh %s", nome);
    }
} Pessoa;

int main(void) {
    Pessoa p;
    p.nome = "Maria";
    p.falar();

    return 0;
}
```

Então vamos manter a suposição de que é possível fazer funções dentro de Structs e voltar para o exemplo dos dados. Como seria uma Struct Dado?

```
typedef struct {
    int valor;
    void sortear(void)
    {
      srand (time(NULL));
      valor = (rand() % 6) + 1;
    }
} Dado;

int main(void) {
    Dado d;          // ←
    d.sortear();     // ←

    return 0;
}
```

Caso fosse possível fazer isso, o nosso programa principal precisaria apenas de duas linhas simples, sempre que quiséssemos jogar um dado.

Em C++ existe um irmão evoluído da Struct, chamada Classe... Na verdade as Classes estão presentes em todas as linguagens orientadas a objetos, não só em C++.

Classes são muito complexas, como ainda vai ser visto neste material, mas por enquanto elas podem ser enxergadas apenas como "Structs que aceitam funções".

A solução do exemplo do Dado, em C++, está no próximo slide. Note que a função main é muito mais clara, e que a mesma classe Dado pode ser reaproveitada em outros programas, sempre que necessário.

```cpp
#include <iostream>
#include <time.h>
using namespace std;

class Dado {
public: // Abstraia deste 'public' por enquanto.
    int valor;
    int sorteia(void) {
        return valor = (rand() % 6) + 1;
    }
    Dado() {
        srand (time(NULL));
        valor = 0;
    }
};

int main(void) {
    Dado dado1,dado2;
    cout << dado1.sorteia() << endl;
    cout << dado2.sorteia() << endl;
    cout << dado1.valor + dado2.valor << endl;
    return 0;
}
```

Algumas nomenclaturas de Orientação a Objetos:

- Uma variável do tipo de uma classe é chamada de Instância ou Objeto. No exemplo anterior, 'dado1' e 'dado2' são instâncias.

- Funções dentro de classes são chamadas de Métodos. 'sorteia()' é um exemplo de método da classe Dado.

- Uma variável dentro de uma classe se chama Propriedade. 'valor' é um exemplo de propriedade de Dado.

- Dentro da classe Dado existe uma função de mesmo nome, Dado(). Ela será o assunto do próximo slide, e se chama Construtor.

Vimos anteriormente, especificamente no slide sobre Structs, o seguinte trecho de código:

```
Pessoa p;
p.nome = "Maria";
p.idade = 30;
p.peso = 75.5;
printf("%s %d %g",p.nome,p.idade,p.peso);
```

Aqui tivemos que primeiro criar a instância do tipo Pessoa, para só depois dar valor a cada propriedade. Agora imagine uma Struct ou Classe com 50 atributos... Seriam 50 linhas de inicialização de variáveis...

Pensando neste problema, existe um método especial, chamado de Construtor. No próximo slide será mostrada uma versão Orientada a Objetos da classe Pessoa.

```cpp
#include <iostream>
using namespace std;

class Pessoa {
    public: // Abstraia do 'public'...
        string nome;
        int idade;
        float peso;

        Pessoa(string nome, int idade, float peso)
        {
            this->nome = nome;
            this-> idade = idade;
            this->peso = peso;
        }
};

int main(void)
{
    Pessoa p("Maria",30,75.5);
    cout << p.nome << endl;
    return 0;
}
```

O Construtor é um método que só pode ser chamado durante a instanciação. Ele serve para *alocar espaço na memória* e para *inicializar atributos*.

No construtor do programa anterior foi usada uma palavra especial, this:

```
Pessoa(string nome, int idade, float peso)
{
    this->nome = nome;
    this-> idade = idade;
    this->peso = peso;
}
```

A linha "this->nome = nome" significa "*O atributo 'nome' presente NESTA classe receberá o valor da variável 'nome' que foi passada por parâmetro.*" Ou seja, o this é sempre referente a um atributo da própria classe em que o Construtor se encontra.

O Construtor pode ser escrito de outra forma, reduzida:

```
Pessoa(string nome, int idade, float peso):
nome(nome), idade(idade), peso(peso) {}
```

Se você quer inicializar um atributo com o valor exato que foi passado por parâmetro, basta usar esta forma reduzida. Note que as inicializações são feitas fora do bloco, fique atento a isso.

A modo padrão e a forma reduzida podem se misturar:

```
Pessoa(string nome, int idade, float peso): nome(nome)
{
    this->idade = idade;
    this->peso = peso;
}
```

Esta inicialização externa não é mais otimizada, e serve apenas para facilitar a visualização do código.

Os parâmetros passados tanto nos métodos quanto no Construtor podem receber valores default. Isto significa que se este parâmetro não for passado, ele receberá um valor pré-definido pelo programador.

```
class Pessoa {
    ...
    Pessoa( string nome , int idade , float peso = 70)
    {
        this->nome = nome;
        this-> idade = idade;
        this->peso = peso;
    }
};

int main(void)
{
    Pessoa p("Maria",30);
    cout << p.peso << endl; // Imprimirá o valor 70.

    return 0;
}
```

Encapsulamento

Os alunos mais atentos já devem ter notado uma falha de segurança grave, presente em todos os códigos em C++ que fiz até agora.
Vejam o código abaixo com atenção e perguntem para si mesmos: “Isto deveria ser possível no meu código?”

```
class Pessoa
{
    public:
        int idade;
        Pessoa(int idade = 0) : idade(idade) {}
};

int main(void)
{
    Pessoa p;
    p.idade = -30;
    cout << p.idade << endl;
    return 0;
}
```

Estou acessando o atributo idade diretamente, e dando valor negativo. O correto não seria proibir isso?

Para resolver este problema existe o Encapsulamento.

- Encapsular é proibir o acesso direto às propriedades. As mudanças que antes eram feitas diretamente (e que permitiam atribuições muitas vezes ilógicas) agora serão feitas por meio de métodos especiais, chamados Getters e Setters.

- Até agora mandei que você abstraísse da palavra 'public', presente em alguns códigos. Existem 3 palavras especiais que definem a visibilidade de uma propriedade ou método dentro da classe: public, private e protected.

- Se algo é listado como public, então ela é acessível através do 'operador ponto', como vim fazendo até agora.

- Se algo é listado como private, ela não é acessível através do ponto.

- A explicação do protected virá em breve...

- Em geral, propriedades são listadas como private e métodos (incluindo o Construtor) são listados como public.

- O código do próximo slide é uma solução para a idade negativa.

```cpp
#include <iostream>
using namespace std;

class Pessoa {
    private:
        // Por convenção, as propriedades devem
        // ser iniciadas por underline.
        // Farei só assim a partir de agora.
        int _idade;
    public:
        int getIdade(void) {
            return _idade;
        }
        void setIdade(int idade) {
            // Prazer, meu nome é Ternário.
            _idade = idade < 0 ? -idade : idade;
        }
        Pessoa(int idade = 0) {
            setIdade(idade);
        }
};

int main(void)
{
    // Tente colocar idade negativa agora...
    Pessoa p(-30);
    cout << p.getIdade() << endl;
    return 0;
}
```

- Se um método não modifica o valor de nenhuma variável, otimize-o chamando de 'const' no fim. Geralmente os métodos Getters podem ser const, pois apenas retornam. Se nenhum valor será modificado, retorne também um const, como no exemplo:

```
const int &getIdade() const {
    return _idade;
}
```

- É possível fazer um só método, que funciona ao mesmo tempo como Get e Set. Quando isto for feito, ele deve ter o nome da propriedade (excetuando o underline) por convenção.

```
int &idade(void) {
    return _idade;
}
```

Este tipo de método impossibilita a solução feita no código anterior, e na prática é como se não houvesse encapsulamento. Ele serve apenas para facilitar uma futura modificação, pois todo o seu código já estaria com "aparência" de Get/Set. #gambiarra

Exemplo com Getter/Setter simultâneos:

```
#include <iostream>
using namespace std;

class Pessoa {
    private:
        int _idade;

    public:
        Pessoa(string nome, int idade, float peso) {
            this->idade() = idade;
        }

        int &idade(void) {
            return _idade;
        }
};

int main(void) {
   Pessoa p("Vitor",22,15.7);

   p.idade() = 23;

   cout << p.idade() << endl;

   return 0;
}
```

Sobrecarga

Uma classe pode ter mais de um método com o mesmo nome, desde que a *quantidade de parâmetros* ou os *tipos dos parâmetros* sejam diferentes.

```
void falar() {
   cout << "Boa noite" << endl;
}

void falar(string frase) {
   cout << "Voce me mandou dizer: " << frase << endl;
}
```

E o mesmo vale para construtores:

```
Pessoa(string nome, int idade) {...}
Pessoa(string nome) {...}
Pessoa(int idade) {...}
Pessoa() {...}
```

Operador ::

Um mesmo código .cpp pode ter várias classes e uma função main ao mesmo tempo, mas este padrão pode complicar a vida do programador em projetos grandes.

Por questões de boa prática de programação e também por convenção da linguagem, o **correto** é colocar apenas uma classe por arquivo. E não num .cpp, mas **.h**

Cada classe deve ser colocada em um arquivo .h de mesmo nome, e seus métodos devem ser escritos separadamente, em um arquivo .cpp.

Para relacionar os métodos num arquivo à classe em outro arquivo, usa-se o operador ::, como é mostrado no próximo slide.

```cpp
// Pessoa.h

#include <iostream>
using namespace std;

class Pessoa
{
   private:
      string _nome;
      int _idade;
   public:
      // O próximo slide explica
      // o que é 'inline'
      inline string &nome(void)
      { return _nome; }
      inline int &idade(void)
      { return _idade; }

      void fazAlgo(void);
      Pessoa(string,int);
      Pessoa(void);
};
```

```cpp
// Pessoa.cpp

#include "Pessoa.h"

Pessoa::Pessoa(void) {
     _nome = "";
     _idade = 18;
}

Pessoa::Pessoa(string nome, int idade)
: _nome(nome) , _idade(idade) {}

void Pessoa::fazAlgo(void) {
   // Imagine várias linhas
   // de código aqui...
}
```

```cpp
// Main.cpp

#include "Pessoa.cpp"

int main(void)
{
     Pessoa p;
     p.nome() = "Maria";
     cout << p.nome() << endl;

     return 0;
}
```

- Se um método possui apenas uma linha, ele deve receber o modificador inline. Se um método é inline, então ele é acessado tão rápido quanto uma variável.

- Métodos inline devem permanecer dentro no .h. Métodos não-inline e Construtores devem ficar no .cpp

- O formato dos métodos escritos no .cpp é:

```
TipoDeRetorno NomeClasse::NomeMetodo(Parametros) {}
```

- Como Construtores não retornam nada, não se escreve nada antes do nome da classe.

- Neste formato, basta uma olhada rápida no arquivo .h e o programador já tem uma boa noção de tudo que a classe faz.

- Dependendo do ambiente em que você estiver, o arquivo a ser incluído na Main.cpp deve ser o .h, e não o .cpp.

Atenção: Qual a diferença entre a Struct Pessoa e a Class Pessoa, representados abaixo?

```
struct Pessoa
{
   int idade;
};

class Pessoa
{
   int idade;
};
```

Embora ambos pareçam ser a mesma coisa neste caso, existe uma diferença sutil. Na struct as propriedades são "public". E na classe as propriedades são "private" por padrão. A idade da struct é acessível, mas a idade da classe não.

Ponteiros em C++

O uso de ponteiros em C++ é diferente do uso de variáveis comuns, e exige alguns cuidados especiais.

Em compensação, eles são muito mais simples do que os ponteiros em C, e facilitam a manipulação de objetos.

Ponteiros se mostrarão muito úteis a partir da segunda parte deste material, que fala sobre Herança e Polimorfismo.

Sinto que você se desesperou ao ver o título deste slide... Então o que acha de uma revisão de ponteiros em C?

Veja dois códigos **em C** nos próximos slides.

```c
#include <stdio.h>

void modifica(int *v1, float *v2) {
    ++*v1;
    ++*v2;
}
int main(void) {
    // Foi criada uma "seta" que aponta para um espaço
    // do tamanho de um int.
    int *i = (int*) malloc(sizeof(int));
    // A seta agora aponta para uma posição de memória com o valor 2
    *i = 2;

    // Outra "seta", que pode apontar para float, mas ainda não aponta
    float *f;
    // Esta é uma variável, já alocada na memória,
    // e que no momento possui um valor arbitrário ("lixo").
    float f2;
    // A posição de memória recebeu um novo valor, 3,14.
    f2 = 3.14;
    // A seta de float agora aponta para a localização
    // da memória em que se encontra o valor 3.14
    f = &f2;

    // Os valores dos ponteiros serão passados como
    // parâmetro de uma função que retorna void,
    // mas terão seus valores modificados mesmo assim.
    modifica(i,f);
    // Execute e veja que os valores impressos são 3 e 4,14.
    printf("%d\n",*i);
    printf("%g\n",*f);

    // O espaço alocado deve ser liberado.
    free(i); free(f);
    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    // Uma "seta" para int, que aponta para uma
    // parte da memória com o tamanho de 3 ints.
    int *i = (int*) malloc(3*sizeof(int));
    // Os valores podem ser acessados como vetores.
    // As posições valem "lixo" de memória.
    printf("%d %d %d\n",i[0],i[1],i[2]);

    // O comando calloc aloca espaço como no malloc,
    // mas inicia os valores com zero.
    int *j = (int*) calloc(3,sizeof(int));
    // Outro modo de acessar posições de ponteiros.
    printf("%d %d %d\n",*j,*(j+1),*(j+2));

    // O comando realloc reaproveita um espaço já alocado
    // e adiciona mais espaço nas posições seguintes.
    int *k = (int*) realloc(j,5*sizeof(int));
    printf("%d %d %d %d %d",k[0],k[1],k[2],k[3],k[4]);

    // O espaço alocado deve ser liberado.
    free(i);
    free(j);
    free(k);

    return 0;
}
```

Resumindo:
- Para alocar espaço, usa-se malloc, calloc ou realloc.
- Com * acessa-se o conteúdo apontado por um ponteiro.
- Com & acessa-se o endereço de uma variável.
- Ponteiros devem ser desalocados quando não forem mais usados, com free.
- As posições podem ser acessadas como vetores comuns.

Em C++ é um pouco diferente. A alocação é feita apenas com uma palavra, new. E a liberação de memória é feita com a palavra delete.

O new chama o Construtor da classe, e o delete chama o Destrutor.

Destrutor é uma função especial, chamada automaticamente no momento em que o programador usa a palavra delete. Esta função desaloca um objeto por conta própria, mas é comum que o próprio programador defina um código dentro desta função, apagando valores de propriedades.

Atenção: delete apaga um ponteiro de uma posição e delete[] apaga um ponteiro de mais de uma posição, como os vistos no programa anterior.

```
#include <iostream>
using namespace std;

class A {
   private:
      int _atrib;

   public:
      inline int &atrib() {
         return _atrib;
      }
   A(int atrib = 0) : _atrib(atrib) {}

   ~A(void)
   {
      cout << "BUM ";
   }
};

int main(void) {
   A a; // Instância comum. É desalocada por conta própria ao fim.
   A *pa; // Ponteiro, ainda não apontando.

   pa = new A; // Agora aponta. Não é desalocado por conta própria.

   delete pa; // Vai desalocar a ponteiro, e escrever "BUM"

   // Ao executar o programa, haverá 2 "BUM", pois o objeto 'a'
   // também é desalocada por ter chegado ao fim do programa.
   return 0;
}
```

Exemplo de uso da palavra delete[], que desaloca um ponteiro de mais de uma posição:

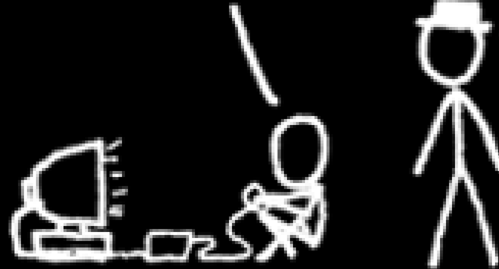
```
#include <iostream>
using namespace std;

int main(void) {

   int *pi = new int[15]; // "calloc(15,sizeof(int))"
   pi[3] = 9;
   cout << pi[3] << endl;
   delete[] pi;

   return 0;
}
```

- Como vimos no slide 36, um ponteiro enviado por parâmetro para uma função pode ter seu valor modificado livremente, e direto na memória, sem precisar de retorno.

- Em C++ podemos mudar o valor de uma variável do mesmo modo, mesmo que ela não seja um ponteiro. Isto se chama **Referência**, e é exemplificado no próximo slide.

```cpp
#include <iostream>
using namespace std;

void mudarValor(int *k) {
    *k = *k + 1;
}

void mudaValorRef(int &k) {
    k = k + 1;
}

int main(void)
{
    int *i;
    int j = 9;
    i = &j;
    mudarValor(i);
    cout << *i << endl;

    // Mesma coisa feita acima, mas sem usar ponteiro.
    int n = 8;
    mudaValorRef(n);
    cout << n << endl;

    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

// É também comum passar como parâmetro um "const ref".
// Assim sabemos que um atributo não será duplicado
// e ao mesmo tempo sabemos que ele não será modificado.

// Funcionalmente, f1() e f2() fazem a mesma coisa.
// Mas internamente, f2() cria uma cópia do parâmetro,
// usa e depois apaga.

int f1(const int &k) {
    return k + 1;
}

int f2(int k) {
    return k + 1;
}

int main(void)
{
    cout << f1(8) << endl;
    cout << f2(8) << endl;

    return 0;
}
```


MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.
つづく