```
#include <bits/stdc++.h>
using namespace std;
```

# The JCC– Holy Bible

Holy grail of programmers

```
g++ -std=c++11 win.cpp
```

01001101 01100001 01111001 00100000 01110100 01101000 01100101
00100000 01100110 01101111 01110010 01100011 01100101 00100000
01100010 01100101 00100000 01110111 01101001 01110100 01101000
00100000 01110101 01110011 00100001

by: Thiago Figueiredo Costa

# Sumário

# 1 Template padrão

```cpp
#include <bits/stdc++.h>
using namespace std;
int main(){
     //code
     return 0;
}
```

Listing 1: Código Padrão C++ 11

```
all: make run

make:
     g++ code.cpp -std=c++11 -o
         program

run: make
     ./program < in
```

Listing 2: Makefile Padrão C++ 11

# 2 Entrada de dados

## 2.1 Inteiros, fracionários e caracteres

Para leitura de uma variável(*var*) separada por delimitador ' ' ou '\t' ou '\n':

```cpp
cin>>var;
```

Listing 3: Leitura com cin

## 2.2 Leitura rápida

Para arquivos muito grandes caso seja necessário uma leitura rápida é recomendável o uso do *scanf* do C, código 4 temos exemplos de leitura e no código 5 leitura até o final do arquivo.

```cpp
int i;          scanf("%d",&i);
long l;         scanf("%ld",&l);
float f;        scanf("%f",&f);
double d;       scanf("%lf",&d);
char c;         scanf("%c",&c);
string s;       scanf("%s",s);
```

Listing 4: Leitura com scanf

```cpp
while(scanf("%d",&x) != EOF){
     //handle
}
```

Listing 5: Leitura com scanf até o final de arquivo

## 2.3 Strings

Para leitura de uma string(*str*) separada por delimitador ' ' ou '\t' ou '\n':

```cpp
cin>>str;
```

Listing 6: Leitura de string com cin

Para leitura de uma linha completa(*str*)(a linha termina com '\n' , mas a string lida não) utiliza-se o comando cin6, entretanto caso haja lixo na memória(geralmente causado após usar um cin6 no final de uma linha) o getline7 não lê nada, por isso devemos usar antes o codigo ignore8:

```
getline(cin,str);
```

Listing 7: Leitura com getline

```
cin.ignore();
```

Listing 8: Limpar o buffer de leitura

### 2.4 Leitura simultânea de valores

Podemos ler uma entrada separada por ' ' ou '\t' contida em uma linha e armazenar em um *vector* usando o código abaixo, considerando que as entradas são do tipo *int*:

```
vector<int> data;
string input;
getline(cin,input);
istringstream iss(input);
copy(istream_iterator<int>(iss), istream_iterator<int>(),
    back_inserter(data));
```

Listing 9: Leitura única de vários dados

### 2.5 End of File

Para ler um arquivo até o fim podemos usar qualquer código acima(3, 6, 7, 9), colocando o comando de leitura dentro de um *loop while*, quando for EOF o *loop* termina, assim como o código 10. Outro exemplo comum é o do código 11 continua lendo até o fim do arquivou ou até achar um número zero.

```
while(cin>>var){
     //handle
}
```

Listing 10: Leitura até o final de arquivo

```
int n;
while(cin>>n,n){
     //handle
}
```

Listing 11: Leitura de inteiros até o final de arquivo ou nulo

## 3 Saída de dados

### 3.1 Precisão de números fracionários

Para imprimir um número com precisão x depois da virgula podemos usar o código 12 ou 13. O comando *fixed* contido nos dois códigos preenche com zeros o número caso a precisão desejada seja maior que o número.

```
cout.precision(x);
cout << fixed << number << endl;
```

Listing 12: Precisão de números fracionários a

```
cout << setprecision(x) << fixed
     << number << endl;
```

Listing 13: Precisão de números fracionários b

### 3.2 Impressão de todos os elementos de um vector

Podemos imprimir todos os elementos de um vector(*vec*) com um separador string(*strsep*) usando uma linha de código(no exemplo é um vetor de inteiros):

```
copy(vec.begin(), vec.end(), ostream_iterator<int>(cout, strsep));
```

Listing 14: Imprimir vector

# 4 Map

Maps são estruturas que associam uma chave de qualquer tipo a um valor de qualquer tipo, com tempo de acesso $O(1)$, muito úteis para dicionários e associações. A ordenação de map pode ser achada em 46

## 4.1 Declaração e uso

A declaração de um map é muito simples, basta informar os tipos da chave e valor, para inserir elementos basta igualar o valor ao map com a chave entre colchetes, para ler basta acessar como um array.

```
key_type key;
value_type value;
map<key_type,value_type> m;
//(v) associando chave ao valor
m[key]=value;
//(v) lendo valor
value_type read=m[key];
```

Listing 15: Inicialização, atribuição e leitura de map

## 4.2 Busca

Podemos saber se o map possui uma determinada chave usando a função de busca16:

```
key_type key;
map<key_type,value_type> m;
if(m.find(key)!=m.end()){
     //possui elemento na chave
}else{
     //chave inexistente
}
```

Listing 16: Busca de chave em map

## 4.3 Set

Set é como se fosse um map apenas com a chave, sem valor associado, a ideia e o uso é o mesmo do map, diferindo apenas na inserção de elementos que acontece através da função *insert*:

```
key_type key;
set<key_type> s;
s.insert(key);
if(s.find(key)!=s.end()){
     //chave definida
}else{
     //chave nao definida
}
```

Listing 17: Uso de set

# 5 Vector

## 5.1 Inicialização

Podemos inicializar um vector vazio ou passando um tamanho($x$), passar um tamanho inicial pode ser bom caso se conheça o limite dele na maiores dos casos pois evita realocação de memória. Um vector pode ter qualquer tipo *type*, o vetor inicializado com um tamanho começa com valor ”0”em seus elementos.

```
1 vector<type> vec;
```

Listing 18: Inicializando vector

```
1 vector<int> vec;
```

Listing 19: Inicializando vector inteiro

```
1 vector<type> vec(x);
```

Listing 20: Inicializando vector com tamanho inicial

Podemos também definir um tamanho para o vetor depois de inicializado com o código abaixo:

```
1 vetor.resize(size);
```

Listing 21: Definindo um tamanho para o vector

### 5.1.1 Cópia de outro vetor

Também é possível inicializar um vetor como uma cópia de outro passando ele como parâmetro:

```
1 vector<type> vec_a;
2 vector<type> vec_b(vec_a);
```

Listing 22: Iniciando um vector com os valores de outro

## 5.2 Manipulação

Podemos acessar acessar valores ja existentes no vector usando [ ] como em 23, ou adicionar/remover um elemento do final do vector usando 24 e 25

```
1 x=vec[i]++;
```

Listing 23: Acessando vector com colchetes

```
1 vec.push_back(x);
```

Listing 24: Adicionando elemento no final do vector

```
1 x=vec.back();
2 vec.pop_back();
```

Listing 25: Removendo elemento do final do vector

## 5.3 Deleção

Podemos deletar elementos de um vector dentro de um range usando o código 26 ou podemos deletar o vector todo usando um dos códigos de 27

```
v.erase(v.begin(),v.begin()+i);
```

Listing 26: Deletando um vector do inicio até i

```
v.erase(v.begin(),v.end());
//ou
v.clear();
```

Listing 27: Deletando o vector todo

### 5.3.1 Deleção de valores específicos

Podemos também deletar do vector apenas os elementos que contém um determinado valor $x$, para isso podemos usar o código 28

```
vec.erase(remove(vec.begin(),vec.end(),x),vec.end());
```

Listing 28: Deletando elementos com valor x

## 5.4 Troca de elementos

Podemos usar o código abaixo para passar o conteúdo do vector $a$ para o vector $b$ e o de vector $b$ para o vector $a$.

```
vetor1.swap(vetor2);
```

Listing 29: Trocar elementos entre vectors

## 5.5 Preencher

Podemos preencher um range do vector usando 30 ou ele todo com 31, usando um valor *value*.

```
fill(v.begin()+i,v.begin()+j,
    value);
```

Listing 30: Preenchendo parte do vector com value

```
fill(v.begin(),v.end(),value);
```

Listing 31: Preencherndo o vector com value

## 5.6 For compacto

Uma forma compacta de escrever um laço *for* para um *vector* é usando o operador ":", um exemplo prático disso é a impressão dos elementos de um *vector*:

```
vector<type> vec;
for(type t:vec)
     cout<<t<<;
cout<<endl;
```

Listing 32: Imprimindo vector com for compacto

# 6 Stack

Uma pilha normal no principio LIFO, pode ser declarada passando o tipo e manipulada usando as chamadas *push, pop, top*, lembrando que a função pop não retorna o valor retirado.

```
type value,value2;
stack<type> s;
s.push(value);
s.push(value2);
type fromstack=s.top();
s.pop();//tira value2 da pilha
```

Listing 33: Pilha

# 7 Queue

Uma fila normal no principio FIFO, pode ser declarada passando o tipo e manipulada usando as chamadas *push, pop, front, back*, lembrando que a função pop não retorna o valor retirado.

```
type value, value2;
queue<type> q;
q.push(value);
q.push(value2);
type fromqueue=s.front();
type fromqueue2=s.back();
s.pop();//tira value da fila
```

Listing 34: Fila

# 8 List

Uma lista normal, pode ser declarada passando o tipo e manipulada usando as chamadas *push_front, pop_front, push_back, pop_back, front, back*, lembrando que a função pop não retorna o valor retirado.

```
type v1,v2,v3;
list<type> l;
l.push_back(v1);
l.push_front(v2);
l.push_back(v3);
type r1=l.front();
type r2=l.back();
l.pop_front();
l.pop_back();
```

Listing 35: Lista

# 9 Math

## 9.1 Truncar

Podemos truncar um número fracionário após a virgula usando cast 36/37 ou podemos truncar *cases* casas depois da vírgula com uma das funções de 38.

```
int x=static_cast<int>value;
```

Listing 36: Trunca o número

```
int x=(int)value;
```

Listing 37: Trunca o número

```cpp
double truncate(double number, int cases){
     return static_cast<int>(number*pow(10,cases))/pow(10,cases);
}

double truncateb(double number, int cases){
     int x=number*pow(10,cases);
     return (double)x/pow(10,cases);
}
```

Listing 38: Trunca o número

## 9.2 Arredondamento para cima ou para baixo

Para arredondar um número fracionário podemos usar as funções do C++ como em 41 e 39, ou podemos usar uma forma mais rápida caso o arredondamento necessário seja após uma divisão como em 40 e 42.

```cpp
double x=ceil(x);
```

Listing 39: Ceil, arredondamento para cima

```cpp
//x=ceil(a/b)
int x=(a+b-1)/b;
//or
int x=((a-1)/b)+1;
```

Listing 40: Ceil div, arredondamento para cima

```cpp
double x=floor(x);
```

Listing 41: Floor, arredondamento para baixo

```cpp
//x=floor(a/b)
int x=((a-1)/b);
```

Listing 42: Floor div, arredondamento para baixo

# 10 Sort

Em casos gerais não é necessário implementar algoritmos de ordenação pois podemos ordenar vectors5 e arrays usando o sort do c++(*quick sort* até que a partição tenha 50 elementos, então *insertion sort*).

## 10.1 std::sort

Para usar o sort padrão do c++ basta fazer como os códigos43 ou 44:

```cpp
vector<string> strs;
sort(strs.begin(), strs.end());
```

Listing 43: Ordenação de vector

```cpp
int values[size];
sort(values, values+size);
```

Listing 44: Ordenação de array

### 10.1.1 Função customizada de comparação

Para ordenar de forma decrescente, ordenar de acordo com vários critérios ou ordenar uma classe customizada basta usar uma função que recebe duas variáveis do tipo e retorna um booleano, a função é o ultimo parâmetro do sort:

```
struct customClass{
     static bool cmp(customClass a, customClass b){
          return a.value==b.value?(
               a.item==b.item?a.name<b.name:a.item<b.item
               ):a.value>=b.value;
     }
     int item;
     double value;
     string name;
};//a funcao cmp nao precisa estar dentro de uma classe
vector<customClass> v;
sort(v.begin(), v.end(),customClass::cmp);
```

Listing 45: Ordenação com vários campos

### 10.1.2 Sort map

Maps4 são automaticamente ordenados pelas chaves, mas caso seja necessário ordenar um mapa pelo seu valor é necessário converte-lo para um vector de pairs e ordenar:

```
bool cmp(pair<key_type,value_type>a, pair<key_type,value_type> b){
     return a.second!=b.second?a.second>b.second:a.first<b.first;
}
map<key_type,value_type> m;
vector<pair<key_type,value_type> > mapvec(m.begin(), m.end());
sort(mapvec.begin(), mapvec.end(), cmp);
```

Listing 46: Ordenação de map

## 10.2 Algoritmos

### 10.2.1 Counting

É o algoritmo mais rápido de todos, estável, $O(n)$, para valores inteiros em um intervalo conhecido.

```
void countingSort(vector<int> v){
     int minValue;
     int maxValue;
     vector<int> count(maxValue-minValue+1);
     vector<int> sort(v);
     //(v)remover se todos elementos forem >=0
          if(minValue<0)
               for(int i=0;i<sort.size();i++)
                    sort[i]+=minValue;

     for(int i=0;i<sort.size();i++)
          count[sort[i]]++;
     for(int i=1;i<count.size();i++)
          count[i]+=count[i-1];
     for(int i=0;i<sort.size();i++){
          v[count[sort[i]]-1]=sort[i];
          //(v)remover se todos elementos forem >=0
               if(minValue<0) v[count[sort[i]]-1]-=minValue;
          count[sort[i]]--;
     }
}
```

Listing 47: Counting sort

### 10.2.2 Quick

É o melhor algoritmo para casos gerais, $O(nlog_n)$, no pior caso é $O(n^2)$, entretanto é possível evitar esse caso usando seleção de pivô por mediana de três.

```
type middle(type a, type b, type c){
     type mid=a;
    if((a<b&&b<c)||(c<b&&b<a))
        mid=b;
     else if((a<c&&c<b)||(b<c&&c<a))
        mid=c;
    return mid;
}

void swap(type &a, type &b){
     type c=a;
     a=b;
     b=c;
}

void quickSort(vector<type> v){
     quickSortR(v,0,v.size()-1);
}

void quickSortR(vector<type> v, int l, int r){
     int i=l,j=r;
     type pivo;
     //(v)pode gerar o pior caso
          pivo=v[(l+r)/2];
     //(v)muito raro de gerar o pior caso
          pivo=middle(v[l],v[(l+r)/2],v[r]);
     while(i<=j){
          while(v[i]<pivo)i++;
          while(v[j]>pivo)j--;
          if(i<=j) swap(v[i++],v[j--]);
     }
     if(l<j)quickSortR(v,l,j);
     if(r>i)quickSortR(v,i,r);
}
```

Listing 48: Quick sort

### 10.2.3 Radix

É rápido e estável, inviável para chaves grandes, $O(nk)$ (k é o comprimento médio das chaves).

```
void radixSort(vector<int> v){
     vector<int> count(10);
     vector<int> sort;
     int max=v[0];
     for(int i=1;i<v.size();i++)
          if(v[i]>max)
               max=v[i];
     for(int exp=1;max/exp>0;exp*=10){//count
          sort=v;
          for(int i=0;i<10;i++)
               count[i]=0;
          for(int i=0;i<sort.size();i++)
               count[(sort[i]/exp)%10]++;
          for(int i=1;i<10;i++)
```

```
                count[i]+=count[i-1];
            for(int i=sort.size()-1;i>=0;i--){
                int at=(sort[i]/exp)%10;
                v[count[at]-1]=sort[i];
                count[at]--;
            }
        }
}
```

Listing 49: Radix sort

### 10.2.4 Merge

Baseado em *divide and conquer*, $O(nlog_n)$, pode consumir muita memória.

```
void mergeSort(vector<type> v){
    mergeSortR(v,0,v.size()-1);
}

void mergeSortR(vector<type> v, int l, int r){
    if(l<r){
        int m=(l+r)/2;
        mergeSortR(v,l,m);
        mergeSortR(v,m+1,r);
        mergeVec(v,l,m,r);
    }
}

void mergeVec(vector<type> v, int l, int m, int r){
    vector<type> aux(r-l+1);
    int idx1=l;
    int idx2=m+1;
    int idx0=0;
    while(1){
        if(idx1<=m&&idx2<=r){
            if(v[idx1]<v[idx2]){
                aux[idx0++]=v[idx1++];
            }else{
                aux[idx0++]=v[idx2++];
            }
        }else{
            while(idx1<=m)
                aux[idx0++]=v[idx1++];
            while(idx2<=r)
                aux[idx0++]=v[idx2++];
            v=aux;
            break;
        }
    }
}
```

Listing 50: Merge sort

### 10.2.5 Heap

O algoritmo com menor variação sempre $O(nlog_n)$, não precisa de memória adicional, bom para arquivos grandes com chaves grandes.

```
void swap(type &a, type &b){
    type c=a;
    a=b;
```

```
     b=c;
}

void buildHeap(vector<type> vec){
     int son;
     for(int i=1;i<=size;i++){
          son=i+1;
          while(son>1&&vec[son/2-1]<vec[son-1]){
               swap(vec[son/2-1],vec[son-1]);
               son/=2;
          }
     }
}

void heapSort(vector<type> v){
     buildHeap(v);
     for(int i=v.size();i>=2;i--){
          swap(v[0],v[i-1]);
          //fixdown
          int father=1,son=2;
          nTtype t=v[0];
          while (son<=(i-1)){
               if(son<(i-1)&&v[son-1]<v[son]) son++;
               if(t>=v[son-1]) break;
               v[father-1]=v[son-1];
               father=son;
               son=2*father;
          }
          v[father-1]=t;
     }
}
```

Listing 51: Heap sort

### 10.2.6 Shell

O mais eficiente entre dos algoritmos quadráticos, bom para entradas de tamanho médio, complexidade desconhecida.

```
void swap(type &a, type &b){
     type c=a;
     a=b;
     b=c;
}
void shellSort(vector<type> v){
     //(v)option 1
          int steep=1;
          while(steep<v.size()) steep=3*steep+1;
          steep/=3;
     //(v)option 2
          int steep=v.size()/2;
     while(steep>0){
          for(int i=0;i<v.size();i++){
               if(i+steep>=v.size())
                    break;
               if(v[i]>v[i+steep]){
                    swap(v[i],v[i+steep]);
                    for(int j=i-1;j>=0;j--)
                         if(v[j]>v[j+steep])
                              swap(v[j],v[j+steep]);
                         else break;
               }
```

```
            }
            //(v)option 1
                steep/=3;
            //(v)option 2
                steep/=2;
        }
}
```

Listing 52: Shell sort

### 10.2.7 Insertion

É o algoritmo mais eficiente para elementos pré-ordenados ou para inserir elementos em um vetor ordenado, complexidade $O(n^2)$.

```
void insertionSort(vector<type> v){
    for(int i=1;i<v.size();i++){
        type pivo=v[i];
        int j=i-1;
        while(j>=0&&pivo<v[j]){
            v[j+1]=v[j];
            j--;
        }
        v[j+1]=pivo;
    }
}
```

Listing 53: Insertion sort

# 11 Characteres

## 11.1 Funções para verificar chars

Todas funções abaixo tem o formato54 e estão na lista 11.1.

```
bool is-----(char in);
```

Listing 54: Formato das funções

As funções retornam verdadeiro se o caractere for:

- **isalnum** uma letra ou número
- **isalpha** uma letra
- **islower** uma letra minúscula
- **isupper** uma letra maiúscula
- **isdigit** um número
- **isxdigit** um número ou letra hexadecimal(a-f, A-F)

- **iscntrl** um caractere de controle(e.g. \n, \t, \0)
- **isgraph** um caractere visível
- **isspace** espaço, tab ou outro caractere de espaçamento
- **isblank** caractere imprimível e não visível(e.g. espaço, tab)
- **isprint** um caractere imprimível
- **ispunct** um caractere de pontuação

## 11.2 Funções para modificar chars

A função *tolower* converte para uma letra minúscula, enquanto *toupper* converte para uma letra maiúscula, ambas retornam o caractere passado como parâmetro caso seja uma letra:

```
1 char c=toupper(c);
```

Listing 55: Passando para letra maiúscula

```
1 char c=tolower(c);
```

Listing 56: Passando para letra minúscula

### 11.3 Tabela ASCII

| Decimal | Binary | Octal | Hex | ASCII | Decimal | Binary | Octal | Hex | ASCII | Decimal | Binary | Octal | Hex | ASCII | Decimal | Binary | Octal | Hex | ASCI |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00000000 | 000 | 00 | NUL | 32 | 00100000 | 040 | 20 | SP | 64 | 01000000 | 100 | 40 | @ | 96 | 01100000 | 140 | 60 | ` |
| 1 | 00000001 | 001 | 01 | SOH | 33 | 00100001 | 041 | 21 | ! | 65 | 01000001 | 101 | 41 | A | 97 | 01100001 | 141 | 61 | a |
| 2 | 00000010 | 002 | 02 | STX | 34 | 00100010 | 042 | 22 | " | 66 | 01000010 | 102 | 42 | B | 98 | 01100010 | 142 | 62 | b |
| 3 | 00000011 | 003 | 03 | ETX | 35 | 00100011 | 043 | 23 | # | 67 | 01000011 | 103 | 43 | C | 99 | 01100011 | 143 | 63 | c |
| 4 | 00000100 | 004 | 04 | EOT | 36 | 00100100 | 044 | 24 | $ | 68 | 01000100 | 104 | 44 | D | 100 | 01100100 | 144 | 64 | d |
| 5 | 00000101 | 005 | 05 | ENQ | 37 | 00100101 | 045 | 25 | % | 69 | 01000101 | 105 | 45 | E | 101 | 01100101 | 145 | 65 | e |
| 6 | 00000110 | 006 | 06 | ACK | 38 | 00100110 | 046 | 26 | & | 70 | 01000110 | 106 | 46 | F | 102 | 01100110 | 146 | 66 | f |
| 7 | 00000111 | 007 | 07 | BEL | 39 | 00100111 | 047 | 27 | ' | 71 | 01000111 | 107 | 47 | G | 103 | 01100111 | 147 | 67 | g |
| 8 | 00001000 | 010 | 08 | BS | 40 | 00101000 | 050 | 28 | ( | 72 | 01001000 | 110 | 48 | H | 104 | 01101000 | 150 | 68 | h |
| 9 | 00001001 | 011 | 09 | HT | 41 | 00101001 | 051 | 29 | ) | 73 | 01001001 | 111 | 49 | I | 105 | 01101001 | 151 | 69 | i |
| 10 | 00001010 | 012 | 0A | LF | 42 | 00101010 | 052 | 2A | * | 74 | 01001010 | 112 | 4A | J | 106 | 01101010 | 152 | 6A | j |
| 11 | 00001011 | 013 | 0B | VT | 43 | 00101011 | 053 | 2B | + | 75 | 01001011 | 113 | 4B | K | 107 | 01101011 | 153 | 6B | k |
| 12 | 00001100 | 014 | 0C | FF | 44 | 00101100 | 054 | 2C | , | 76 | 01001100 | 114 | 4C | L | 108 | 01101100 | 154 | 6C | l |
| 13 | 00001101 | 015 | 0D | CR | 45 | 00101101 | 055 | 2D | - | 77 | 01001101 | 115 | 4D | M | 109 | 01101101 | 155 | 6D | m |
| 14 | 00001110 | 016 | 0E | SO | 46 | 00101110 | 056 | 2E | . | 78 | 01001110 | 116 | 4E | N | 110 | 01101110 | 156 | 6E | n |
| 15 | 00001111 | 017 | 0F | SI | 47 | 00101111 | 057 | 2F | / | 79 | 01001111 | 117 | 4F | O | 111 | 01101111 | 157 | 6F | o |
| 16 | 00010000 | 020 | 10 | DLE | 48 | 00110000 | 060 | 30 | 0 | 80 | 01010000 | 120 | 50 | P | 112 | 01110000 | 160 | 70 | p |
| 17 | 00010001 | 021 | 11 | DC1 | 49 | 00110001 | 061 | 31 | 1 | 81 | 01010001 | 121 | 51 | Q | 113 | 01110001 | 161 | 71 | q |
| 18 | 00010010 | 022 | 12 | DC2 | 50 | 00110010 | 062 | 32 | 2 | 82 | 01010010 | 122 | 52 | R | 114 | 01110010 | 162 | 72 | r |
| 19 | 00010011 | 023 | 13 | DC3 | 51 | 00110011 | 063 | 33 | 3 | 83 | 01010011 | 123 | 53 | S | 115 | 01110011 | 163 | 73 | s |
| 20 | 00010100 | 024 | 14 | DC4 | 52 | 00110100 | 064 | 34 | 4 | 84 | 01010100 | 124 | 54 | T | 116 | 01110100 | 164 | 74 | t |
| 21 | 00010101 | 025 | 15 | NAK | 53 | 00110101 | 065 | 35 | 5 | 85 | 01010101 | 125 | 55 | U | 117 | 01110101 | 165 | 75 | u |
| 22 | 00010110 | 026 | 16 | SYN | 54 | 00110110 | 066 | 36 | 6 | 86 | 01010110 | 126 | 56 | V | 118 | 01110110 | 166 | 76 | v |
| 23 | 00010111 | 027 | 17 | ETB | 55 | 00110111 | 067 | 37 | 7 | 87 | 01010111 | 127 | 57 | W | 119 | 01110111 | 167 | 77 | w |
| 24 | 00011000 | 030 | 18 | CAN | 56 | 00111000 | 070 | 38 | 8 | 88 | 01011000 | 130 | 58 | X | 120 | 01111000 | 170 | 78 | x |
| 25 | 00011001 | 031 | 19 | EM | 57 | 00111001 | 071 | 39 | 9 | 89 | 01011001 | 131 | 59 | Y | 121 | 01111001 | 171 | 79 | y |
| 26 | 00011010 | 032 | 1A | SUB | 58 | 00111010 | 072 | 3A | : | 90 | 01011010 | 132 | 5A | Z | 122 | 01111010 | 172 | 7A | z |
| 27 | 00011011 | 033 | 1B | ESC | 59 | 00111011 | 073 | 3B | ; | 91 | 01011011 | 133 | 5B | [ | 123 | 01111011 | 173 | 7B | { |
| 28 | 00011100 | 034 | 1C | FS | 60 | 00111100 | 074 | 3C | < | 92 | 01011100 | 134 | 5C | \ | 124 | 01111100 | 174 | 7C | \| |
| 29 | 00011101 | 035 | 1D | GS | 61 | 00111101 | 075 | 3D | = | 93 | 01011101 | 135 | 5D | ] | 125 | 01111101 | 175 | 7D | } |
| 30 | 00011110 | 036 | 1E | RS | 62 | 00111110 | 076 | 3E | > | 94 | 01011110 | 136 | 5E | ^ | 126 | 01111110 | 176 | 7E | ~ |
| 31 | 00011111 | 037 | 1F | US | 63 | 00111111 | 077 | 3F | ? | 95 | 01011111 | 137 | 5F | _ | 127 | 01111111 | 177 | 7F | DEL |

Figura 1: Tabela ASCII

# 12 Strings

## 12.1 Substring

Para gerar uma substring de uma string(*str*) usamos o código 57, ou, 58, onde *start* é a posição inicial da string e *size* é o tamanho da substring.

```
1 str.substr(start,size);
```

Listing 57: Gerando substring

```
1 str.substr(size);
```

Listing 58: Gerando substring

## 12.2 Remover caracteres da string

Para remover todas as ocorrências de um caractere($c$) de uma string(*str*) usamos o comando 59

```
1 str.erase(remove(str.begin(),str.end(),c),str.end());
```

Listing 59: Remover sequencia de caracteres

## 12.3 Inverter String

Para inverter uma string(*str*) usamos o comando 60

```
reverse(str.begin(), str.end());
```

Listing 60: Inverter String

## 12.4 Achar uma string

Para achar uma string(*str2*) em uma string(*str*) podemos usar find, que retorna um size_t a posição onde achou a *str2*, a posição de inicio da busca(*init*) pode ser passada como parâmetro(Caso não seja passado como parâmetro a busca começa na posição 0). Podemos por exemplo armazenar onde achamos a string como no caso 61 ou verificar se achamos como em 62:

```
size_t founded;
founded=str.find(str2,init);
```

Listing 61: Buscando uma string a partir da posição init

```
if(str.find(str2)!=string::npos)
    //achei
else
    //nao achei
```

Listing 62: Verificando se contem uma string

## 12.5 Substituir todas as ocorrências

Para substituir todas as ocorrências de uma *substring* em uma string podemo usar o código abaixo:

```
void replaceAll(string &s,const string &search,const string &
    replace){
    for(size_t pos=0;;pos+=replace.length()){
        pos=s.find(search,pos);
        if(pos==string::npos) break;
        s.erase(pos,search.length());
        s.insert(pos,replace);
    }
}
```

Listing 63: Substituindo todas as ocorrências

## 12.6 Converter para string

### 12.6.1 Converção formatada

Pode ser necessário converter um número com determinada precisão, base ou transformar em uma string mais complexa, para isso podemos criar uma string formatada usando:

```
char buff[size_buffer];//definir o tamanho de acordo com a string
snprintf(buff,size_buffer,"%.02f %d %c %X",f,i,c,hex);
string str=string(buff);//acima foi apenas um exemplo
```

Listing 64: Criando uma string formatada

### 12.6.2 Converção simples

Podemos converter qualquer variável numérica do C++ para string usando o código abaixo:

```
1 string s=to_string(number);
```

Listing 65: Números reais para string

## 12.7 Converter de string

Podemos transformar uma string em qualquer base numérica para uma variável numérica de qualquer tipo do c++, as funções tem o formato66 e estão na lista 12.7.

```
1 string str;
2 //(v) base 10
3 type number=sto--(str);
4 //(v) base customizada: 8,10,16,...
5 string::size_type sz;
6 type number=sto--(str,&sz,16);
```

Listing 66: Converter string para número real

- **stoi** converte para um int(inteiro)
- **stol** converte para long
- **stoul** converte para unsigned long
- **stoll** converte para long long
- **stoull** converte para unsigned long long
- **stof** converte para float
- **stod** converte para double
- **stold** converte para long double

# 13 Paradigmas

## 13.1 Mochila

Um paradigma comum em programação é o problema da mochila, a entrada são objetos que tem um peso e um valor, e há um peso máximo de objetos que se pode carregar, qual é o valor máximo de todos os objetos que se pode carregar? Para responder de problemas desse tipo usamos programação dinâmica como abaixo:

```
int bag[nOfInputs+1][capacity+1];
int weights[nOfInputs];
int values[nOfInputs];
for(int i=0;i<=nOfInputs;i++)
     bag[i][0]=0;
for(int j=0;j<=capacity;j++)
     bag[0][j]=0;
for(int i=1;i<=nOfInputs;i++){
     for(int j=1;j<=capacity;j++){
          if(weights[i-1]<=j)
            bag[i][j]=max((values[i-1] + bag[i-1][j-weights[i-1]]),
                 bag[i-1][j]);
         else
             bag[i][j]=bag[i-1][j];
     }
}
int values_max=bag[nOfInputs][capacity];
```

Listing 67: Problema da mochila boleana com pesos

# 14 Grafos

## 14.1 Representação

## 14.2 Busca

### 14.2.1 Largura

### 14.2.2 Profundidade

## 14.3 Representação