UNIVERSIDAD DE COSTA RICA

PROGRAMACIÓN COMPILADA: C++
SEGUNDA PARTE

Prof. Marlon Brenes y Prof. Federico Muñoz
Escuela de Física, Universidad de Costa Rica

EFis Escuela de Física

Manejo de excepciones

- Las excepciones son un método para recuperar el programa de situaciones problemáticas

```cpp
#include <iostream>

double division(int a, int b){

  if(b == 0){
    throw "Division by zero condition!";
  }

  return (a / b); // Aquí está ocurriendo un cast interno de tipos de variables
}

int main(){

  int x = 50;
  int y = 0;
  double z = 0;

  try{
    z = division(x, y);
    std::cout << z << std::endl;
  }catch(const char* msg){ // const es una expresión que denota que "algo" es constante y no se puede cambiar
                           // dentro de la función. La función catch toma una hilera de caracteres constantes
                           // como variable
    std::cerr << msg << std::endl;
  }

  return 0;
}
```

- excepciones.cpp

Manejo de excepciones

- Existen excepciones standard
  - e.g., bad_alloc, bad_cast

- Las excepciones tienen alto costo computacional
  - Incluso cuando la excepción nunca ocurre!
  - Por esta razón es mejor evitarlas en lugares donde la eficiencia del programa es crítica

La biblioteca standard

- La *standard template library (STL)* es una biblioteca de clases contenedoras, algoritmos e iteradores
  - Provee muchos de los algoritmos y estructuras de datos básicos comúnmente utilizados en las ciencias de la computación

- Existe una ventaja ENORME al usar la biblioteca standard en lugar de implementar nuestros propios algoritmos
  - Dentro de muchas, podemos nombrar seguridad de memoria
    - Por ejemplo, la biblioteca standard nos permite usar una estructura de datos para representar una matriz
    - Podemos hacer esto sin usar punteros y asignación dinámica de memoria, dado que ocurre de manera interna

- La biblioteca standard va a ser una entidad siempre presente en nuestras aplicaciones científicas
  - Gran parte de los algoritmos y estructuras hoy en día se pueden accesar mediante `<iostream>`

La biblioteca standard

- Casi todos los componentes del STL son plantillas *(templates)*
  - Más de esto más adelante

- STL se accesa mediante el namespace `std`

http://www.cplusplus.com/reference/stl/

Contenedores

- Los contenedores son clases (más de esto más adelante) cuyo propósito es mantener otros objetos en memoria
- Algunas de las clases incluídas:
  - vector
  - list
  - deque *(double-ended queue)*
  - set, multiset
  - map, multimap
  - hash_set, hash_multiset, hash_map (encriptación)

std::vector

- Es un contenedor sequencial que representa un arreglo que puede cambiar en tamaño
- **Sus elementos son contiguos en memoria** (asignados en el heap)
- Conoce su propio tamaño `vector::size()` y sus cualidades se almacenan dentro del objeto en si
- Los elementos se pueden añadir de forma dinámica `vector::push_back(T)`
- Para garantizar contigüidad en memoria, cada inserción tiene un costo computacional (reasignación)
- Para evitar reasignación de memoria se puede reservar espacio `vector::reserve(int)`
- Reservación de espacio no cambia el tamaño `vector::size()`, pero si cambia su capacidad `vector::capacity()`
- `vector::operator[]` no provee checks de acceso
- `vector::at()` provee checks de acceso (excepción + terminación)

http://www.cplusplus.com/reference/vector/vector/

std::vector

```cpp
#include <iostream>
#include <vector> // Puede no ser necesario, depende de la biblioteca instalada

int main(){

  std::vector<double> a;

  a.push_back(7.0);
  std::cout << "a: size " << a.size() << " capacity " << a.capacity() << std::endl;

  a.push_back(8.0);
  std::cout << "a: size " << a.size() << " capacity " << a.capacity() << std::endl;

  std::vector<double> b;
  b.reserve(8);
  std::cout << "b: size " << b.size() << " capacity " << b.capacity() << std::endl;

  std::vector<double> c(5); // constructor con tamaño en inicialización
  std::cout << "c: size " << c.size() << " capacity " << c.capacity() << std::endl;

  return 0;
}
```

- vector.cpp

EFis Escuela de Física

Matriz como un vector de vectores

- Una matrix se puede definir como un vector de vectores

```cpp
#include <iostream>
#include <vector>

int main(){

  std::vector< std::vector<int> > vec; // Se necesitan espacios entre los <>

  for(int i = 0; i < 10; ++i){
    std::vector<int> row; // Crea una fila vacía
    for(int j = 0; j < 20; ++j){
      row.push_back(i * j); // Añade un elemento a la fila
    }
    vec.push_back(row); // Añade la fila al vector principal (matriz)
  }

  // También se puede declarar en construcción
  // Igual se puede modificar su tamaño
  std::vector< std::vector<int> > vec_2(4, std::vector<int>(4));

  return 0;
}
```

- El problema es que ahora los elementos no son contiguos en memoria (solamente contiguos a nivel de filas, mas no columnas)
  - `matrix_vecvec.cpp`

Contigüidad

- La contigüidad en memoria de elementos de una matrix es sumamente importante
  - Permite realizar optimizaciones de operaciones lineales que no serían posibles de otra forma
  - Permite accesos con comportamiento determinado
  - Permite el uso de bibliotecas especializadas de álgebra lineal

- En este curso, NUNCA se usará una representación de objetos de álgebra lineal con elementos no contiguos en memoria

Matriz *(the right way)*

```cpp
#include <iostream>
#include <vector>

int main(){

  int rows = 4; // Número de filas
  int cols = 3; // Número de columnas

  // La matrix va a ser un arreglo contiguo en memoria
  // i.e., un arreglo 1-dimensional con rows*cols elementos
  std::vector<double> matrix(rows * cols, 0.0);
  for(int i = 0; i < rows; ++i){
    for(int j = 0; j < cols; ++j){
      matrix[(i * cols) + j] = (i * cols) + j; // Ojo con este acceso
    }
  }

  // Podemos visualizar la matrix de la siguiente forma
  for(int i = 0; i < rows; ++i){
    for(int j = 0; j < cols; ++j){
      std::cout << matrix[(i * cols) + j] << " ";
    }
    std::cout << std::endl;
  }

  return 0;
}
```

El acceso ahora es un poco más complicado (con índices) pero los elementos están contiguos en memoria

- matrix_contigua.cpp

std::map

- Los mapas son *contenedores asociativos* que almacenas elementos conformados por una combinación de un *valor clave* y un *valor objeto*
  - Las parejas clave-valor pueden ser añadidas, modificadas y eliminadas
  - La ventaja es que los mapas permiten clasificaciones más óptimas
  - El método de clasificación se puede establecer en construcción

```cpp
#include <iostream>
#include <map>

int main(){

  std::map<std::string, std::string> phonebook; // Notese los argumentos de plantilla
  std::cout << phonebook.size() << std::endl;

  phonebook["Marlon"] = "2345-6789"; // Entrada no existe, se crea
  std::cout << phonebook.size() << std::endl;
  phonebook["Marlon"] = "9876-5432"; // Entrada existe, se modifica
  std::cout << phonebook.size() << std::endl;

  std::cout << phonebook["Marlon"] << std::endl;
  std::cout << phonebook["Fede"] << std::endl; // Entrada en blanco

  return 0;
}
```

- `mapa.cpp`

Iteradores

- Un iterador es un objeto que apunta a algún elemento de algún contenedor
  - Pueden iterar a través de los elementos usando operadores
  - Ejemplos de operadores son incremento (++) y dereferencia (*)
  - Los punteros son un tipo de iterador

```cpp
#include <iostream>
#include <iterator>

int main(){

  std::vector<double> a(5, 0.0);
  for(int i = 0; i < 5; ++i) a[i] = static_cast<double>(i); // Conversión de tipos

  std::vector<double>::iterator myIt;
  for(myIt = a.begin(); myIt != a.end(); ++myIt)
  // El puntero se asigna al inicio y se itera hasta que se llega al final
    std::cout << *myIt << std::endl; // Dereferenciación
  std::cout << std::endl;

  a.erase(a.begin(), a.begin() + 2); // Borramos un rango del arreglo
  for(myIt = a.begin(); myIt != a.end(); ++myIt)
    std::cout << *myIt << std::endl;
  std::cout << std::endl;

  myIt = a.begin() + 1;
  std::cout << *(myIt + 2) << std::endl;
  std::cout << myIt[2] << std::endl;

  return 0;
}
```

- `iterador.cpp`

Iteradores

- Los contenedores de la STL tienen al menos un iterador `begin()` y un `end()`
- El iterador end() apunta al elemento después del final, así que no debe ser dereferenciado
- Algunos contenedores como std::list solo proveen iteradores bidireccionales (i.e., `myIt++` y `myIt--`) pero no de acceso aleatorio (`myIt+3` no es permitido)
- Los iteradores no hacen checks de frontera: se pueden accesar elementos no asignados

- `iterador.cpp`

Algoritmos

- Esta sección contiene una colección de funciones diseñadas especialmente para ser usadas en rangos de elementos dentro de un contenedor
  - e.g., `find, sort, min, max, for_each`

Algoritmos

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

void print(int i){
  std::cout << " " << i;
}

int main(){

  std::vector<int> a;
  for(int i = 0; i < 5; ++i) a.push_back(5 - i);

  std::vector<int>::iterator it;
  it = std::find(a.begin(), a.end(), 3); // El rango de búsqueda y el elemento

  if( it != a.end())
    std::cout << "Element found in vector: " << *it << std::endl;
  else
    std::cout << "Element not found" << std::endl;

  // Recordar que end() no debe ser dereferenciado
  for(it = a.begin(); it < a.end(); ++it) std::cout << *it << " ";
  std::cout << std::endl;

  // Clasificación parcial
  std::sort(a.begin(), a.begin() + 3);
  for(it = a.begin(); it < a.end(); ++it) std::cout << *it << " ";
  std::cout << std::endl;

  // for_each
  std::cout << "Printing with for_each" << std::endl;
  for_each(a.begin(), a.end(), print);
  std::cout << std::endl;

  return 0;
}
```

- algoritmos.cpp

std::complex

- La funcionalidad de números complejos en la STL es una clase de plantillas (más de esto más adelante)
- `std::complex<float>, std::complex<double>`
- Se pueden usar funciones de la STL en números complejos: `exp, log, cos, sin, sqrt`, etc.

```cpp
#include <complex>
#include <iostream>

int main(){

  std::complex<double> a(3.0, 5.0); // <-- Esto invoca al constructor
  std::complex<double> b = a * a;

  std::cout << "a = " << a << " a.real() = " << a.real() << " a.imag() = " << a.imag() << std::endl;
  std::cout << "b = " << b << std::endl;

  std::cout << " exp(b) = " << std::exp(b) << std::endl;

  return 0;
}
```

- `complejos.cpp`

Muchas más cosas!

- Generadores de números aleatorios `<random>`
- Soporte para multithreading (no se usará en este curso)
- *Tuples* `<tuple>`
- ...

- `iterador.cpp`

http://www.cplusplus.com/reference/stl/

Laboratorio

- **Completar** `matmault.cpp`