



UNIVERSIDAD DE COSTA RICA

PROGRAMACIÓN COMPILADA: C++ SEGUNDA PARTE

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

Manejo de excepciones

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

```
1 #include <iostream>
2
3 double division(int a, int b){
4
5     if(b == 0){
6         throw "Division by zero condition!";
7     }
8
9     return (a / b); // Aquí está ocurriendo un cast interno de tipos de variables
10 }
11
12 int main(){
13
14     int x = 50;
15     int y = 0;
16     double z = 0;
17
18     try{
19         z = division(x, y);
20         std::cout << z << std::endl;
21     }catch(const char* msg){ // const es una expresión que denota que "algo" es constante y no se puede cambiar
22                             // dentro de la función. La función catch toma una hilera de caracteres constantes
23                             // como variable
24         std::cerr << msg << std::endl;
25     }
26
27     return 0;
28 }
```

- 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 acceder 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 incluidas:
 - ↳
 - vector
 - list
 - deque (*double-ended queue*)
 - set, multiset
 - map, multimap
 - hash_set, hash_multiset, hash_map (encriptación)

std::vector

- Es un contenedor secuencial 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)

std::vector

```
1 #include <iostream>
2 #include <vector> // Puede no ser necesario, depende de la biblioteca instalada
3
4 int main(){
5
6     std::vector<double> a;
7
8     a.push_back(7.0);
9     std::cout << "a: size " << a.size() << " capacity " << a.capacity() << std::endl;
10
11    a.push_back(8.0);
12    std::cout << "a: size " << a.size() << " capacity " << a.capacity() << std::endl;
13
14    std::vector<double> b;
15    b.reserve(8);
16    std::cout << "b: size " << b.size() << " capacity " << b.capacity() << std::endl;
17
18    std::vector<double> c(5); // constructor con tamaño en inicialización
19    std::cout << "c: size " << c.size() << " capacity " << c.capacity() << std::endl;
20
21    return 0;
22 }
```

- vector.cpp

Matriz como un vector de vectores

- Una matrix se puede definir como un vector de vectores

```
1 #include <iostream>
2 #include <vector>
3
4 int main(){
5
6     std::vector< std::vector<int> > vec; // Se necesitan espacios entre los <>
7
8     for(int i = 0; i < 4; ++i){
9         std::vector<int> row; // Crea una fila vacía
10        for(int j = 0; j < 20; ++j){
11            row.push_back(i * j); // Añade un elemento a la fila
12        }
13        vec.push_back(row); // Añade la fila al vector principal (matriz)
14    }
15
16    // También se puede declarar en construcción
17    // Igual se puede modificar su tamaño
18    std::vector< std::vector<int> > vec_2(4, std::vector<int>(4));
19
20    return 0;
21 }
```

- 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*)

```
1 #include <iostream>
2 #include <vector>
3
4 int main(){
5
6     int rows = 4; // Número de filas
7     int cols = 3; // Número de columnas
8
9     // La matrix va a ser un arreglo contiguo en memoria
10    // i.e., un arreglo 1-dimensional con rows*cols elementos
11    std::vector<double> matrix(rows * cols, 0.0);
12    for(int i = 0; i < rows; ++i){
13        for(int j = 0; j < cols; ++j){
14            matrix[(i * cols) + j] = (i * cols) + j; // Ojo con este acceso
15        }
16    }
17
18    // Podemos visualizar la matrix de la siguiente forma
19    for(int i = 0; i < rows; ++i){
20        for(int j = 0; j < cols; ++j){
21            std::cout << matrix[(i * cols) + j] << " ";
22        }
23        std::cout << std::endl;
24    }
25
26    return 0;
27 }
```

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

```
1 #include <iostream>
2 #include <map>
3
4 int main(){
5
6     std::map<std::string, std::string> phonebook; // Notese los argumentos de plantilla
7     std::cout << phonebook.size() << std::endl;
8
9     phonebook["Marlon"] = "2345-6789"; // Entrada no existe, se crea
10    std::cout << phonebook.size() << std::endl;
11    phonebook["Marlon"] = "9876-5432"; // Entrada existe, se modifica
12    std::cout << phonebook.size() << std::endl;
13
14    std::cout << phonebook["Marlon"] << std::endl;
15    std::cout << phonebook["Fede"] << std::endl; // Entrada en blanco
16
17    return 0;
18 }
```

- 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

```
1 #include <iostream>
2 #include <iterator>
3
4 int main(){
5
6     std::vector<double> a(5, 0.0);
7     for(int i = 0; i < 5; ++i) a[i] = static_cast<double>(i); // Conversión de tipos
8
9     std::vector<double>::iterator myIt;
10    for(myIt = a.begin(); myIt != a.end(); ++myIt)
11        // El puntero se asigna al inicio y se itera hasta que se llega al final
12        std::cout << *myIt << std::endl; // Dereferenciación
13    std::cout << std::endl;
14
15    a.erase(a.begin(), a.begin() + 2); // Borramos un rango del arreglo
16    for(myIt = a.begin(); myIt != a.end(); ++myIt)
17        std::cout << *myIt << std::endl;
18    std::cout << std::endl;
19
20    myIt = a.begin() + 1;
21    std::cout << *(myIt + 2) << std::endl;
22    std::cout << myIt[2] << std::endl;
23
24    return 0;
25 }
```

- `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 acceder 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

```
1 #include <iostream>
2 #include <algorithm>
3 #include <vector>
4
5 void print(int i){
6     std::cout << " " << i;
7 }
8
9 int main(){
10
11     std::vector<int> a;
12     for(int i = 0; i < 5; ++i) a.push_back(5 - i);
13
14     std::vector<int>::iterator it;
15     it = std::find(a.begin(), a.end(), 3); // El rango de búsqueda y el elemento
16
17     if( it != a.end())
18         std::cout << "Element found in vector: " << *it << std::endl;
19     else
20         std::cout << "Element not found" << std::endl;
21
22     // Recordar que end() no debe ser dereferenciado
23     for(it = a.begin(); it < a.end(); ++it) std::cout << *it << " ";
24     std::cout << std::endl;
25
26     // Clasificación parcial
27     std::sort(a.begin(), a.begin() + 3);
28     for(it = a.begin(); it < a.end(); ++it) std::cout << *it << " ";
29     std::cout << std::endl;
30
31     // for_each
32     std::cout << "Printing with for_each" << std::endl;
33     for_each(a.begin(), a.end(), print);
34     std::cout << std::endl;
35
36     return 0;
37 }
```

- 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.

```
1 #include <complex>
2 #include <iostream>
3
4 int main(){
5
6     std::complex<double> a(3.0, 5.0); // <-- Esto invoca al constructor
7     std::complex<double> b = a * a;
8
9     std::cout << "a = " << a << " a.real() = " << a.real() << " a.imag() = " << a.imag() << std::endl;
10    std::cout << "b = " << b << std::endl;
11
12    std::cout << " exp(b) = " << std::exp(b) << std::endl;
13
14    return 0;
15 }
```

• `complejos.cpp`

Muchas más cosas!

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

- `iterador.cpp`

Nuestra primera aplicación científica en C++

- └─→ • Ver demostración en el archivo `pi.cpp`

Nuestra segunda aplicación científica en C++

- └─→ • Ver demostración en el archivo `matmault.cpp`