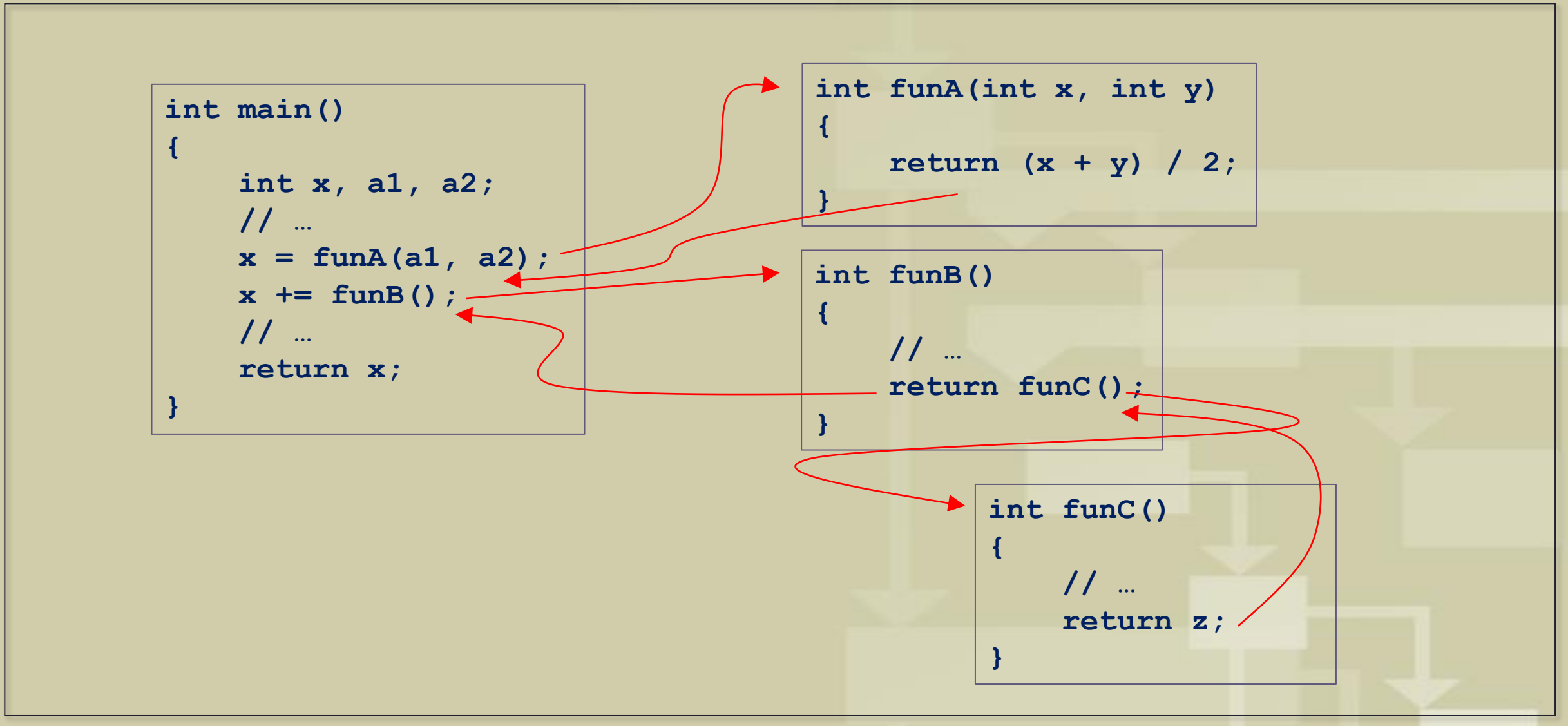
Qué vimos?

- Flujo de ejecución de un programa
  - La función **`main`** toma el control del programa, ejecutando todas las sentencias que contiene su cuerpo.
  - La ejecución de una función es secuencial, pero pueden incluirse bloques condicionales, repetitivos (iteraciones) o llamados a funciones (inclusive a ella misma).
  - Llamar a una función es transferirle el control de ejecución. Cuando la función termina, el control vuelve a la sentencia posterior a la llamada original.
  - Una función termina cuando llega al final de su cuerpo, o alcanza una sentencia **`return`**.

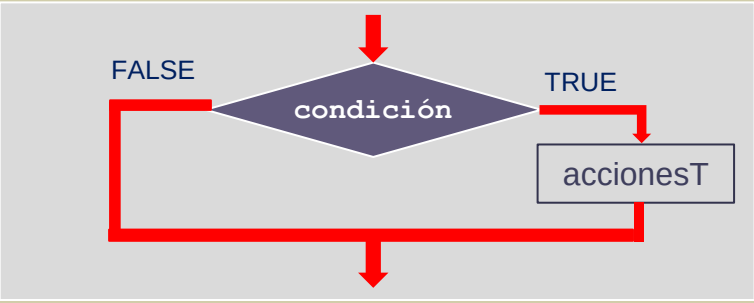
Flujo de ejecución de un programa



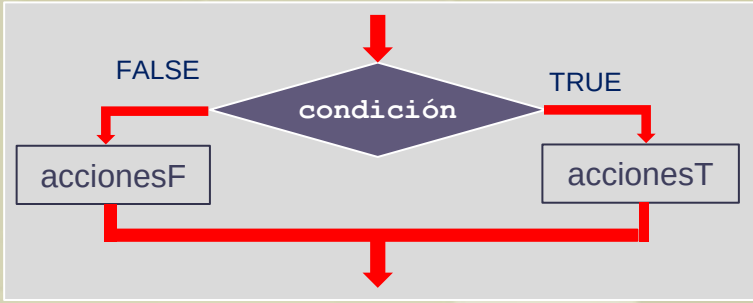
Bloque condicional `if`

- El bloque **`if`** es una estructura de control condicional. Su sintaxis es:

```
if( condición ) {
    accionT1;
    accionT2;
}
```




```
if( condición ) {
    accionT1;
    accionT2;
} else {
    accionF1;
    accionF2;
}
```



Condición lógica

- **`condición`** es una expresión que establece la condición lógica que debe satisfacerse para que el cuerpo de la estructura de control se ejecute.

- La expresión puede contener operadores relacionales, lógicos, algebraicos y de bits.

- En el caso de que la expresión termine en un valor numérico, este se tratará como verdadero si su valor es distinto de 0, y como falso si su valor es 0.

Bloque repetitivo **`while`**

- El bloque **`while`** es una estructura de control repetitivo. Su sintaxis es:

```
while( condición ) {
    accion1;
    accion2;
    // …
}
```

- En el caso de que el cuerpo posea una sola sentencia puede omitirse las `{ }`. Esto vale para cualquier estructura de control de flujo.

Bloque Repetitivo **`for`**

- El bloque **`for`** es una estructura de control repetitivo. Su sintaxis es:

```
for( a; b; c ) {
    accion1;
    accion2;
}
```

- La expresión **`a`** es la inicialización, y puede tener una, ninguna o varias sentencias separados por ‘**`,`**’. Se ejecuta una única vez antes de la iteración.
- La expresión **`b`** establece la condición de ejecución de las acciones de la iteración, se evalúa antes de ejecutarlas.
- La expresión **`c`** es ejecutada al final de cada ciclo, antes de reevaluar la condición.

Bloque Repetitivo **do-while**

- El bloque **do-while** es una estructura de control repetitivo levemente diferente al **while**. Su sintaxis es:

```
do {
     accion1;
     accion2;
} while( condición );
```

- Las acciones se ejecutan al menos una vez antes de evaluar la condición.
- Notar el ';' después del **while(condición)**.

Bloque Repetitivo `for` para secuencias

- Esta variante de **`for`** (range-**`for`**) es utilizada para iterar de manera simple sobre todos los elementos de secuencias que naturalmente tienen un inicio y un fin, un rango. Su sintaxis es:

```
vector<int> v = { 5, 6, 9, 4, 6, 8 };

for( int x : v ) {          // Para cada x en v
    cout << x << '\n';
}
```

- Para iteraciones más complejas, como mirar un elemento de cada 3 o sólo los elementos en la primera mitad de la secuencia, se utiliza el **`for`** más general y tradicional.

Sentencia **break**

- La sentencia **break** produce la salida inmediata del **while**, **for**, **do-while** ó **switch** en que se encuentra. Por ejemplo:

```
while( 1 ) {
    double x;
    cin >> x;
    if( x < 0.0 )       /* si x es negativo */
        break;          /* salgo del while */
    cout << sqrt(x) << "\n";
}
/* ... y vengo a parar acá */
cout << "El número es negativo!!\n";
```

Sentencia **`continue`**

- La sentencia **`continue`** termina la iteración actual del **`while`**, **`for`** ó **`do-while`** en que se encuentra y vuelve al comienzo de la iteración. Por ejemplo:

```
for( int i = 0; i < 20; ++i ) {
    if( i == 13 )
        continue;      /* supersticioso yo? */
    cout << i << '\n';
}
```

Sentencia goto

- La sentencia **goto** produce un salto incondicional a una sentencia con una etiquetada que se encuentra en algún lugar de la misma función. Por ejemplo:

```
  for( int i = 0; i < 100; ++i ) {
      for( int j = 0; j < 100; ++j ) {
          if( i * j == i + j )
              goto afuera;
          cout << i << ' ' << j << '\n';
      }
  }

afuera:
  cout << "Salida de emergencia??\n";
```

Manejo de errores

- Cuando programamos, debemos convivir con diversos tipos de errores. Son inevitables completamente.
- No todos los errores se pueden eliminar.
- Vamos a asumir que nuestros programas:
  1. Deberían producir resultados correctos para todas las entradas válidas (input data).
  2. Deberían dar mensajes de error razonables para todas las entradas inválidas.
  3. No necesitan preocuparse por errores de hardware.
  4. No necesitan preocuparse por errores del sistema operativo.
  5. Tienen permitido terminar su ejecución después de encontrar un error.

Errores de runtime

- Resueltos los errores de compilación y de link, obtenemos un programa que se puede ejecutar. Acá el código del programa debe detectar y manejar los errores.
- Si una función detecta un error, debe informarlo a la función que la llamó. Esto lleva a:
  - Código de detección de errores en las funciones (por ejemplo de validez de argumentos) que deben ser reportados a quien llamó a la función.
  - Código de manipulación de errores reportados por funciones llamadas.

```
int area(int a, int b) {
   if( a < 0 || b < 0 )   // checkeo
      return -1:          // retorna error
    return a * b;
}
```

```
int f(int a, int b) {
     int s = area(a - 3, b – 4);
     if( s == -1 )           // checkeo
          error("area: bad args\n");
     return s;
}
```

- Manejar adecuadamente los errores es una tarea compleja y tediosa. Hacer un correcto manejo de errores puede llegar a ser 75% del código. Es probable que se introduzcan nuevos errores en el código de detección y manejo de errores...

Aserciones

- Una aserción es una condición que se supone verdadera y que debería siempre cumplirse para poder seguir ejecutando un algoritmo. C++ permite realizar aserciones a través de la función **`assert(condición)`** no teniendo ningún efecto si la condición se cumple o interrumpiendo el programa con un mensaje alusivo si la condición no se cumple.
- Es un mecanismo primitivo que se suele utilizar en etapas tempranas de desarrollo y/o depuración. Sin efecto en modo “release”.

ejem4_6.cpp

```
/* ejemplo de assert */
#include "icom_helpers.h"
int main()
{
    cout << "Ingrese un número positivo ";
    double v;
    cin >> v;
    assert(v >= 0);
    cout << "la raiz cuadrada es: " << sqrt(v) << '\n';
    return 0;
}
```