Metodología de la Programación

Tema 3. Funciones (ampliación)

Andrés Cano Utrera
(acu@decsai.ugr.es)
Departamento de Ciencias de la Computación e I.A.

Curso 2019-2020

Contenido del tema

Contenido del tema

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - Valor de retorno: El int devuelto por main informa al SO sobre el posible código de error del programa.
  - Argumentos de main:

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - Valor de retorno: Entero devuelto por `main` informando al SO sobre el posible código de error del programa.
  - Argumentos de `main`:

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - Valor de retorno: Entero devuelto por `main` informando al SO sobre el posible código de error del programa.
  - Argumentos de `main`:

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - **Valor de retorno**: El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error
  - **Argumentos de main**:
    - `int argc`: Número de argumentos pasados al ejecutar el programa.
    - `char *argv[]`: Array de cadenas con cada uno de los argumentos.
      - `argv[0]`: Nombre del ejecutable
      - `argv[1]`: Primer argumento

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - **Valor de retorno**: El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error
  - **Argumentos de main**:
    - `int argc`: Número de argumentos pasados al ejecutar el programa.
    - `char *argv[]`: Array de cadenas con cada uno de los argumentos.
      - `argv[0]`: Nombre del ejecutable
      - `argv[1]`: Primer argumento

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - **Valor de retorno**: El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error
  - **Argumentos de main**:
    - `int argc`: Número de argumentos usados al ejecutar el programa.
    - `char *argv[]`: Array de cadenas con cada uno de los argumentos.
      - `argv[0]`: Nombre del ejecutable
      - `argv[1]`: Primer argumento

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - **Valor de retorno**: El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error
  - **Argumentos de main**:
    - `int argc`: Número de argumentos usados al ejecutar el programa.
    - `char *argv[]`: Array de cadenas con cada uno de los argumentos.
      - `argv[0]`: Nombre del ejecutable
      - `argv[1]`: Primer argumento

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - **Valor de retorno**: El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error
  - **Argumentos de main:**
    - `int argc`: Número de argumentos usados al ejecutar el programa.
    - `char *argv[]`: Array de cadenas con cada uno de los argumentos.
      - `argv[0]`: Nombre del ejecutable
      - `argv[1]`: Primer argumento

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - **Valor de retorno**: El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error
  - **Argumentos de main:**
    - `int argc`: Número de argumentos usados al ejecutar el programa.
    - `char *argv[]`: Array de cadenas con cada uno de los argumentos.
      - `argv[0]`: Nombre del ejecutable
      - `argv[1]`: Primer argumento

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - **Valor de retorno**: El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error
  - **Argumentos de main**:
    - `int argc`: Número de argumentos usados al ejecutar el programa.
    - `char *argv[]`: Array de cadenas con cada uno de los argumentos.
      `argv[0]`: Nombre del ejecutable
      `argv[1]`: Primer argumento
      ...

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - **Valor de retorno**: El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error
  - **Argumentos de main**:
    - `int argc`: Número de argumentos usados al ejecutar el programa.
    - `char *argv[]`: Array de cadenas con cada uno de los argumentos.
      `argv[0]`: Nombre del ejecutable
      `argv[1]`: Primer argumento
      ...

La función main I

- Un programa C++ comienza cuando el SO transfiere el control a `main` y finaliza cuando esta función acaba.
- Hasta ahora, hemos usado la siguiente cabecera simple para `main`:
  `int main()`
- C++ permite una versión ampliada de la cabecera de `main`:
  `int main(int argc, char *argv[])`
  - **Valor de retorno**: El `int` devuelto por `main` informa al SO sobre el posible código de error del programa.
    - 0: Ok (valor por defecto)
    - Otro valor: Algún tipo de error
  - **Argumentos de main**:
    - `int argc`: Número de argumentos usados al ejecutar el programa.
    - `char *argv[]`: Array de cadenas con cada uno de los argumentos.
      `argv[0]`: Nombre del ejecutable
      `argv[1]`: Primer argumento
      ...

La función main II: Ejemplo

```
 1 #include <iostream>
 2 using namespace std;
 3 int main(int argc, char *argv[]){
 4     if (argc<3){
 5         cerr << "Uso: "
 6             << " <Fichero1> <Fichero2> ..."<<endl;
 7         return 1;
 8     }
 9     else{
10         cout<<"Numero argumentos: "<<argc<<endl;
11         for (int i=0; i<argc; ++i){
12             cout<<argv[i]<<endl;
13         }
14     }
15     return 0;
16 }
```

La función main III

Podemos convertir las cadenas estilo C al tipo `string`

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main(int argc, char *argv[])
5  {
6      string par;
7      cout<<"Argumentos: "<<endl;
8      for (int i=0; i<argc; ++i)
9      {
10         par=argv[i];
11         cout<<par<<endl;
12     }
13     return 0;
14 }
```

Contenido del tema

Referencias

Referencia

Es una especie de alias a otro dato u objeto. Se usa en:

- Paso de parámetros por referencia en una función o método
- Referencias como alias a otras variables
- Devolución por referencia desde una función

Referencias como alias a otras variables

Referencias como alias a otras variables

Una variable referencia es un alias a otra variable:

`<tipo> & <identificador> = <iniciador> ;`

Las variables referencia deben **inicializarse** en su declaración y **no pueden reasignarse** como alias a otras variables.

- **Ejemplo 1**:

```
int a=0;
int &ref=a;
ref=5;
cout<<a<<endl;
```

- **Ejemplo 2**:

```
int v[5]={1,2,3,4,5};
int &ref=v[3];
ref=0;
cout<<v[3]<<endl;
```

Referencias como alias a otras variables

Referencias como alias a otras variables

Una variable referencia es un alias a otra variable:

`<tipo> & <identificador> = <iniciador> ;`

Las variables referencia deben **inicializarse** en su declaración y **no pueden reasignarse** como alias a otras variables.

- **Ejemplo 1**:

```
int a=0;
int &ref=a;
ref=5;
cout<<a<<endl;
```

- **Ejemplo 2**:

```
int v[5]={1,2,3,4,5};
int &ref=v[3];
ref=0;
cout<<v[3]<<endl;
```

Referencias como alias a otras variables

Referencias como alias a otras variables

Una variable referencia es un alias a otra variable:

`<tipo> & <identificador> = <iniciador> ;`

Las variables referencia deben **inicializarse** en su declaración y **no pueden reasignarse** como alias a otras variables.

- **Ejemplo 1**:

```
int a=0;
int &ref=a;
ref=5;
cout<<a<<endl;
```

- **Ejemplo 2**:

```
int v[5]={1,2,3,4,5};
int &ref=v[3];
ref=0;
cout<<v[3]<<endl;
```

Paso de parámetros por referencia en una función o método

Por referencia o variable

- Debe usarse un **lvalue** en el parámetro actual (en la llamada a la función o método).
- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos.
- Una modificación en el parámetro formal, conlleva la misma modificación en el parámetro actual.

Paso de parámetros por referencia en una función o método

Por referencia o variable

- Debe usarse un **lvalue** en el parámetro actual (en la llamada a la función o método).
- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos.
- Una modificación en el parámetro formal, conlleva la misma modificación en el parámetro actual.

Paso de parámetros por referencia en una función o método

Por referencia o variable

- Debe usarse un **lvalue** en el parámetro actual (en la llamada a la función o método).
- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos.
- Una modificación en el parámetro formal, conlleva la misma modificación en el parámetro actual.

Paso de parámetros por referencia en una función o método

Ejemplo:

Función que intercambia el valor de dos variables

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

void Swap(char &c1, char &c2){
  char aux=c1;
  c1=c2;
  c2=aux;
}
int main(){
  char a='a', b='b';

  Swap(a,b);
  cout << "a=" << a
       << " y b=" << b << endl;
}
```

Paso de parámetros por referencia constante

Paso por referencia constante

Habitualmente se usa para pasar objetos de gran tamaño que no van a modificarse en la función o método

```
double calcularMedia(const VectorSD& v){
    double suma = 0.0;
    for(int i=0; i< v.nElementos(); i++){
        suma += v.getDato(i);
    }
    return suma/v.nElementos();
}
int main(){
    VectorSD miVector;
    for(int i = 0; i < 1000000; i++)
        miVector.aniadir(uniforme(0,50));
    cout << calcularMedia(miVector);
}
```

Paso de parámetros por referencia en una función o método

Paso por referencia constante: ¿se puede llamar con un rvalue?

Podemos usar un lvalue y también una expresión (un **rvalue**) para llamar a una función o método que espera un argumento por referencia constante.

```
void mostrar(const double& dato){
    cout << "Dato: " << dato << endl;
}
int main(){
    int a = 3.0;
    mostrar(a);   // Llamada con un lvalue
    mostrar(a+2); // ¿Es válida esta llamada?
}
```

Llamada a funciones o métodos con lvalues o rvalues

Llamada a funciones con parámetros actuales lvalue o rvalue

Según sea el parámetro formal, podremos llamar a la función o método con parámetros actuales lvalue o rvalue.

- **Paso por valor**: argumento actual puede ser una expresión, una constante o una variable.
- **Paso por referencia**: argumento actual solo puede ser un lvalue.
- **Paso por referencia constante**: argumento actual puede ser una expresión, una constante o una variable.

Llamada a funciones o métodos con lvalues o rvalues

Llamada a funciones con parámetros actuales lvalue o rvalue

Según sea el parámetro formal, podremos llamar a la función o método con parámetros actuales lvalue o rvalue.

- **Paso por valor**: argumento actual puede ser una expresión, una constante o una variable.
- **Paso por referencia**: argumento actual solo puede ser un lvalue.
- **Paso por referencia constante**: argumento actual puede ser una expresión, una constante o una variable.

Llamada a funciones o métodos con lvalues o rvalues

Llamada a funciones con parámetros actuales lvalue o rvalue

Según sea el parámetro formal, podremos llamar a la función o método con parámetros actuales lvalue o rvalue.

- **Paso por valor**: argumento actual puede ser una expresión, una constante o una variable.
- **Paso por referencia**: argumento actual solo puede ser un lvalue.
- **Paso por referencia constante**: argumento actual puede ser una expresión, una constante o una variable.

Devolución por referencia

Función con devolución por referencia

Una función puede devolver una referencia a un dato u objeto

```
int& valor(int *v, int i){
    return v[i];
}
```

La referencia puede usarse en el lado derecho de una asignación

```
int main(){
    int v[]={3,5,2,7,6};
    int a=valor(v,3);
}
```

Pero también en el lado izquierdo de la asignación

```
int main(){
    int v[]={3,5,2,7,6};
    valor(v,3)=0;
}
```

Devolución por referencia

Función con devolución por referencia

Una función puede devolver una referencia a un dato u objeto

```
int& valor(int *v, int i){
    return v[i];
}
```

La referencia puede usarse en el lado derecho de una asignación

```
int main(){
    int v[]={3,5,2,7,6};
    int a=valor(v,3);
}
```

Pero también en el lado izquierdo de la asignación

```
int main(){
    int v[]={3,5,2,7,6};
    valor(v,3)=0;
}
```

Devolución por referencia

Función con devolución por referencia

Una función puede devolver una referencia a un dato u objeto

```
int& valor(int *v, int i){
    return v[i];
}
```

La referencia puede usarse en el lado derecho de una asignación

```
int main(){
    int v[]={3,5,2,7,6};
    int a=valor(v,3);
}
```

Pero también en el lado izquierdo de la asignación

```
int main(){
    int v[]={3,5,2,7,6};
    valor(v,3)=0;
}
```

Devolución por referencia

Devolución de referencias a datos locales

La devolución de referencias a datos locales a una función es un error típico: Los datos locales se destruyen al terminar la función.

```
#include <iostream>
using namespace std;
int& funcion()
{
    int x=3;
    return x; //Aviso al compilar: devolución referencia a variable local
}
int main()
{
    int y=funcion(); // Error de ejecución
    cout << y << endl;
}
```

Devolución por referencia

Devolución de referencia constante

Una función puede devolver una referencia constante: significa que el dato referenciado es constante.

```
const int &valor(const int *v, int i){
    return v[i];
}
int main(){
    int v[3]={0,1,2};
    v[2]=3*5; // Correcto
    valor(v,2)=3*5 // Error compilación, pues la referencia es const
    int res=valor(v,2)*3; // Correcto
}
```

Devolución por referencia

Devolución de puntero constante

Lo mismo ocurre cuando una función devuelve un puntero: podemos hacer que éste sea `const`: significa que el dato apuntado es constante.

```
const int *valor(int *v, int i){
    return v+i;
}
int main(){
    int v[3];
    v[2]=3*5; // Correcto
    *(valor(v,2))=3*5; // Error, pues el puntero devuelto es const
    int res=*(valor(v,2))*3; // Correcto
}
```

Contenido del tema

Parámetros con valor por defecto

Parámetros con valor por defecto

Una función o método puede tener parámetros con un valor por defecto

- Deben ser los últimos de la función.
- En la llamada a la función, si solo se especifican un subconjunto de ellos, deben ser los primeros.

```
void funcion(char c, int i=7){
    ...
}
int main(){
    funcion('a',8);
    funcion('z');
}
```

Parámetros con valor por defecto

Parámetros con valor por defecto

Una función o método puede tener parámetros con un valor por defecto

- Deben ser los últimos de la función.
- En la llamada a la función, si solo se especifican un subconjunto de ellos, deben ser los primeros.

```
void funcion(char c, int i=7){
    ...
}
int main(){
    funcion('a',8);
    funcion('z');
}
```

Parámetros con valor por defecto: Ejemplo

```
#include <iostream>
using namespace std;
int volumenCaja(int largo=1, int ancho=1, int alto=1);
int main()
{
    cout << "Volumen por defecto: " << volumenCaja() << endl;
    cout << "El volumen de una caja (10,1,1) es: " <<
            volumenCaja(10) << endl;
    cout << "El volumen de una caja (10,5,1) es: " <<
            volumenCaja(10,5) << endl;
    cout << "El volumen de una caja (10,5,2) es: " <<
            volumenCaja(10,5,2) << endl;
    return 0;
}
int volumenCaja( int largo, int ancho, int alto )
{
    return largo * ancho * alto;
}
```

Contenido del tema

Sobrecarga de funciones

Sobrecarga de funciones

C++ permite definir varias funciones en el mismo ámbito con el mismo nombre. C++ selecciona la función adecuada en base al número, tipo y orden de los argumentos.

```
void funcion(int x){
   ...
}
void funcion(double x){
   ...
}
void funcion(char *c){
   ...
}
void funcion(int x, double y){
   ...
}
```

```
int main(){
    char *c;
    funcion(3);
    funcion(4.5);
    funcion(4,9.3);
    funcion(c);
}
```

Sobrecarga de funciones

Conversión implícita de tipos

C++ puede aplicar conversión implícita de tipos para buscar la función adecuada.

```
void funcion(double x){
  cout << "double" << x << endl;
}
void funcion(char *p){
  cout << "char *" << *p <<endl;
}
int main(){
    funcion(4.5);
    funcion(3); // conversión implícita
}
```

Sobrecarga de funciones

Distinción por el tipo devuelto

C++ no puede distinguir entre dos versiones de función que solo se diferencian en el tipo devuelto.

```
int funcion(int x){
  return x*2;
}
double funcion(int x){
  return x/3.0;
}
int main(){
    int x=funcion(3);
    double f=funcion(5);
}
```

Sobrecarga de funciones

Uso de const en punteros y referencias

C++ puede distinguir entre versiones en que un parámetro puntero o bien referencia es const en una versión y en la otra no.

```
#include <iostream>
using namespace std;
void funcion(double &x){
  cout << "funcion(double &x): " << x <<endl;
}
void funcion(const double &x){
  cout << "funcion(const double &x): " << x <<endl;
}
int main(){
    double x=2;
    const double A=4.5;
    funcion(A);
    funcion(x);
}
```

Sobrecarga de funciones

Uso de const en punteros y referencias

C++ puede distinguir entre versiones en que un parámetro puntero o bien referencia es const en una versión y en la otra no.

```
#include <iostream>
using namespace std;
void funcion(double *p){
  cout << "funcion(double *p): " << *p <<endl;
}
void funcion(const double *p){
  cout << "funcion(const double *p): " << *p <<endl;
}
int main(){
    double x=2;
    const double A=4.5;
    funcion(&A);
    funcion(&x);
}
```

Sobrecarga de funciones

Uso de const en parámetros por valor

Sin embargo, C++ no puede distinguir entre versiones en que un parámetro por valor es const en una versión y en la otra no.

```
#include <iostream>
using namespace std;
void funcion(double x){
  cout << "funcion(double x): " << x <<endl;
}
void funcion(const double x){
  cout << "funcion(const double x): " << x <<endl;
}
int main(){
    double x=2;
    const double A=4.5;
    funcion(A);
    funcion(x);
}
```

Sobrecarga de funciones

Ambigüedad

A veces pueden darse errores de ambigüedad

```
void funcion(int a, int b){
   ...
}
void funcion(double a, double b){
   ...
}
int main(){
    funcion(2,4);
    funcion(3.5,4.2);
    funcion(2,4.2); //Ambiguo
    funcion(3.5,4); //Ambiguo
    funcion(3.5,static_cast<double>(4));
}
```

Sobrecarga de funciones

Otro ejemplo de ambigüedad

En este caso al usar funciones con parámetros por defecto

```
void funcion(char c, int i=7){
    ...
}
void funcion(char c){
    ...
}
int main(){
    funcion('a',8);
    funcion('z');
}
```

Contenido del tema

Funciones inline

Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.
- Código generado de mayor tamaño.
- El compilador puede que no haga caso al calificador `inline`.
- Suelen colocarse en ficheros de cabecera (`.h`) ya que el compilador necesita su definición para poder expandirlas.

Funciones inline

Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.
- Código generado de mayor tamaño.
- El compilador puede que no haga caso al calificador `inline`.
- Suelen colocarse en ficheros de cabecera (`.h`) ya que el compilador necesita su definición para poder expandirlas.

Funciones inline

Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.
- Código generado de mayor tamaño.
- El compilador puede que no haga caso al calificador `inline`.
- Suelen colocarse en ficheros de cabecera (`.h`) ya que el compilador necesita su definición para poder expandirlas.

Funciones inline

Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.
- Código generado de mayor tamaño.
- El compilador puede que no haga caso al calificador `inline`.
- Suelen colocarse en ficheros de cabecera (`.h`) ya que el compilador necesita su definición para poder expandirlas.

Funciones inline

Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.
- Código generado de mayor tamaño.
- El compilador puede que no haga caso al calificador `inline`.
- Suelen colocarse en ficheros de cabecera (`.h`) ya que el compilador necesita su definición para poder expandirlas.

Funciones inline

Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.
- Código generado de mayor tamaño.
- El compilador puede que no haga caso al calificador `inline`.
- Suelen colocarse en ficheros de cabecera (`.h`) ya que el compilador necesita su definición para poder expandirlas.

Funciones inline

Función inline

Es una forma de declarar una función para que el compilador genere una copia de su código, cada vez que es llamada, para evitar una llamada a función, y así aumentar la velocidad de ejecución del programa.

- Se definen colocando `inline` antes del tipo de retorno en la definición de la función.
- Suelen ser funciones pequeñas y que son llamadas con mucha frecuencia.
- Fueron introducidas en C++ para solucionar los problemas de las macros (no comprobación de tipos, problemas al expandirlas, etc).
- Ejecución más rápida en general.
- Código generado de mayor tamaño.
- El compilador puede que no haga caso al calificador `inline`.
- Suelen colocarse en ficheros de cabecera (`.h`) ya que el compilador necesita su definición para poder expandirlas.

Funciones inline: Ejemplo

```
1 #include <iostream>
2 inline bool numeroPar(const int n){
3     return (n%2==0);
4 }
5 int main(){
6     std::string parimpar;
7     parimpar=numeroPar(25)?"par":"impar";
8     std::cout<<"Es 25 par?: " << parimpar;
9 }
```

Contenido del tema

Variables locales static

Variable local static

Es una variable local de una función o método que no se destruye al acabar la función, y que mantiene su valor entre llamadas.

- Se inicializa la primera vez que se llama a la función.
- Conserva el valor anterior en sucesivas llamadas a la función.
- Es obligatorio asignarles un valor en su declaración.

```
#include <iostream>
double cuadrado(double numero){
    static int contadorLlamadas=1;
    std::cout<<"Llamadas a cuadrado: "
             <<contadorLlamadas<<std::endl;
    contadorLlamadas++;
    return numero*numero;
}
int main(){
    for(int i=0; i<10; ++i)
        std::cout<<i<<"^2 = "<<cuadrado(i)<<std::endl;
}
```

Variables locales static

Variable local static

Es una variable local de una función o método que no se destruye al acabar la función, y que mantiene su valor entre llamadas.

- Se inicializa la primera vez que se llama a la función.
- Conserva el valor anterior en sucesivas llamadas a la función.
- Es obligatorio asignarles un valor en su declaración.

```
#include <iostream>
double cuadrado(double numero){
    static int contadorLlamadas=1;
    std::cout<<"Llamadas a cuadrado: "
             <<contadorLlamadas<<std::endl;
    contadorLlamadas++;
    return numero*numero;
}
int main(){
    for(int i=0; i<10; ++i)
        std::cout<<i<<"^2 = "<<cuadrado(i)<<std::endl;
}
```

Variables locales static

Variable local static

Es una variable local de una función o método que no se destruye al acabar la función, y que mantiene su valor entre llamadas.

- Se inicializa la primera vez que se llama a la función.
- Conserva el valor anterior en sucesivas llamadas a la función.
- Es obligatorio asignarles un valor en su declaración.

```
#include <iostream>
double cuadrado(double numero){
    static int contadorLlamadas=1;
    std::cout<<"Llamadas a cuadrado: "
             <<contadorLlamadas<<std::endl;
    contadorLlamadas++;
    return numero*numero;
}
int main(){
    for(int i=0; i<10; ++i)
        std::cout<<i<<"^2 = "<<cuadrado(i)<<std::endl;
}
```

Contenido del tema

Contenido del tema

Introducción a la recursividad

Recursividad

Un problema podrá resolverse de forma recursiva si podemos expresar su solución en términos de un conjunto de datos o entrada de menor tamaño.

Factorial de un número

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Solución recursiva de un problema

Como podemos ver, para resolver un problema recursivamente es necesario definir cómo resolver:

- Los casos base: casos del problema que se resuelven sin recursividad.
- Los casos generales: cuando el problema es suficientemente grande o complejo, la solución se expresa de forma recursiva.

Introducción a la recursividad

Recursividad

Un problema podrá resolverse de forma recursiva si podemos expresar su solución en términos de un conjunto de datos o entrada de menor tamaño.

Factorial de un número

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Solución recursiva de un problema

Como podemos ver, para resolver un problema recursivamente es necesario definir cómo resolver:

- Los casos base: casos del problema que se resuelven sin recursividad.
- Los casos generales: cuando el problema es suficientemente grande o complejo, la solución se expresa de forma recursiva.

Introducción a la recursividad

Recursividad

Un problema podrá resolverse de forma recursiva si podemos expresar su solución en términos de un conjunto de datos o entrada de menor tamaño.

Factorial de un número

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Solución recursiva de un problema

Como podemos ver, para resolver un problema recursivamente es necesario definir cómo resolver:

- Los casos base: casos del problema que se resuelven sin recursividad.
- Los casos generales: cuando el problema es suficientemente grande o complejo, la solución se expresa de forma recursiva.

Contenido del tema

Ejemplos de funciones recursivas

Factorial de un número

- Implementación secuencial:

```
int factorial(int n){
   int res = 1;
   for(int i=2; i<=n; ++i)
      res *= i;
   return res;
}
```

- Implementación recursiva:

```
int factorial(int n){
   if (n==0)
      return 1;
   else
      return n * factorial(n-1);
}
```

Ejemplos de funciones recursivas

Factorial de un número

- Implementación secuencial:

```
int factorial(int n){
   int res = 1;
   for(int i=2; i<=n; ++i)
      res *= i;
   return res;
}
```

- Implementación recursiva:

```
int factorial(int n){
   if (n==0)
      return 1;
   else
      return n * factorial(n-1);
}
```

Ejemplos de funciones recursivas

Suma de los elementos de un array

Podemos resolverlo recursivamente sumando dos cantidades:

- La primera, la suma de los elementos desde el 0 hasta elemento central.
- La segunda, las suma de los elementos desde el central al último.

```
int suma(const int array[], int desde, int hasta){
   if(desde==hasta)
      return array[desde];
   else{
      int centro = (desde+hasta)/2;
      return suma(array, desde, centro) + suma(array, centro+1, hasta);
   }
}
```

Ejemplos de funciones recursivas

Suma de los elementos de un array

Podemos resolverlo recursivamente sumando dos cantidades:

- La primera, la suma de los elementos desde el 0 hasta elemento central.
- La segunda, las suma de los elementos desde el central al último.

```
int suma(const int array[], int desde, int hasta){
   if(desde==hasta)
      return array[desde];
   else{
      int centro = (desde+hasta)/2;
      return suma(array, desde, centro) + suma(array, centro+1, hasta);
   }
}
```

Ejemplos de funciones recursivas

Mostrar al revés los elementos de un array

Mostramos al revés los elementos del subarray derecho (todos los elementos menos el primero), y luego el primer elemento.

```
void mostrarAlReves(const int array[], int utiles, int desde){
   if(desde<utiles) { // Si hay elementos a mostrar
      mostrarAlReves(array, utiles, desde+1);
      cout << array[desde] << endl;
   }
}
```

Ejemplos de funciones recursivas

Mostrar al revés los elementos de un array

Mostramos al revés los elementos del subarray derecho (todos los elementos menos el primero), y luego el primer elemento.

```
void mostrarAlReves(const int array[], int utiles, int desde){
   if(desde<utiles) { // Si hay elementos a mostrar
      mostrarAlReves(array, utiles, desde+1);
      cout << array[desde] << endl;
   }
}
```

Contenido del tema

Recursivo versus iterativo

Solución con algoritmo recursivo versus iterativo

En muchos casos, una solución recursiva requiere **más recursos de tiempo y memoria** que una solución iterativa.

Por ello, la recursividad tiene su gran aplicación para **problemas complejos**, cuya solución recursiva es más fácil de obtener, más estructurada y sencilla de mantener.

Recursivo versus iterativo

Solución con algoritmo recursivo versus iterativo

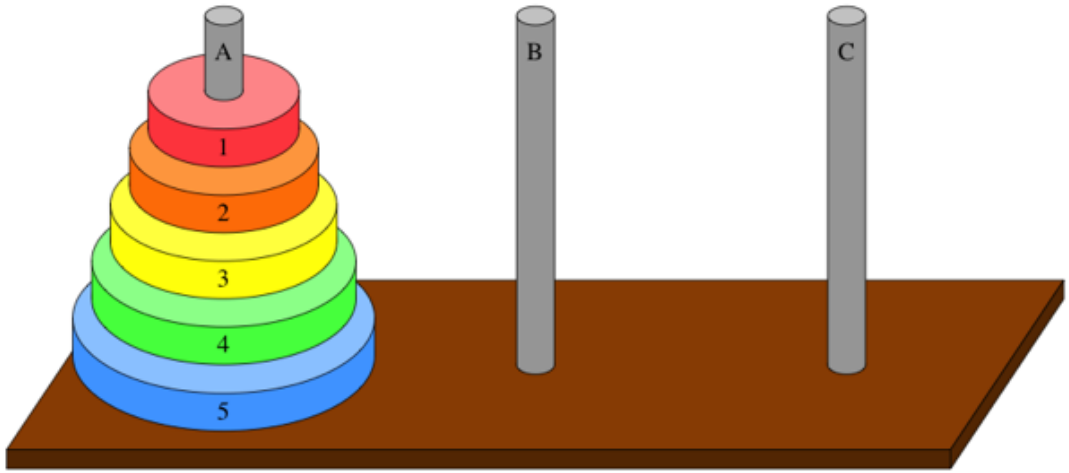
En muchos casos, una solución recursiva requiere **más recursos de tiempo y memoria** que una solución iterativa.
Por ello, la recursividad tiene su gran aplicación para **problemas complejos**, cuya solución recursiva es más fácil de obtener, más estructurada y sencilla de mantener.

Problema de las torres de Hanoi

Problema de las torres de Hanoi

Tenemos un conjunto de tres torres.

- La primera tiene apiladas n fichas de mayor a menor tamaño.
- El problema consiste en pasar las n fichas de la primera a la tercera torre teniendo en cuenta
  - En cada paso solo se puede mover una ficha.
  - Una ficha de un tamaño no puede apilarse sobre otra de menor tamaño.

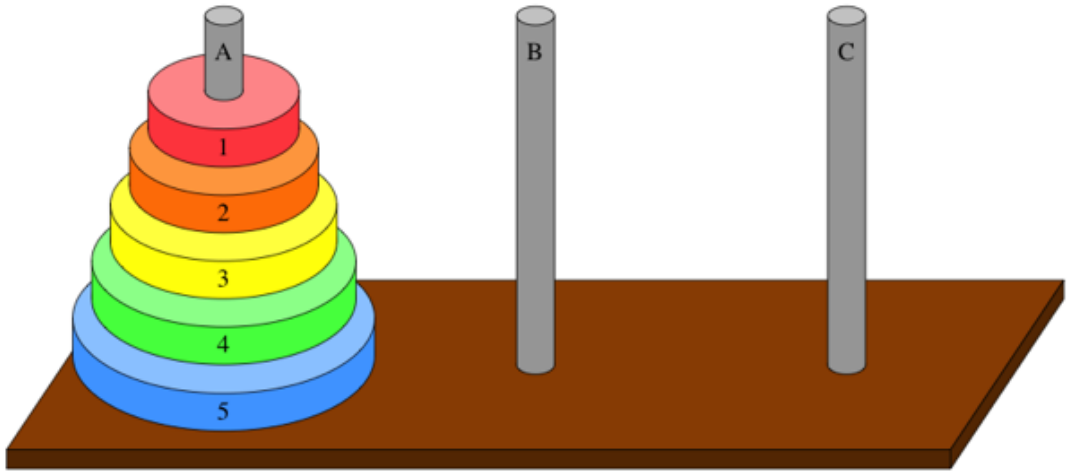


Problema de las torres de Hanoi

Problema de las torres de Hanoi

Tenemos un conjunto de tres torres.

- La primera tiene apiladas n fichas de mayor a menor tamaño.
- El problema consiste en pasar las n fichas de la primera a la tercera torre teniendo en cuenta
  - En cada paso solo se puede mover una ficha.
  - Una ficha de un tamaño no puede apilarse sobre otra de menor tamaño.

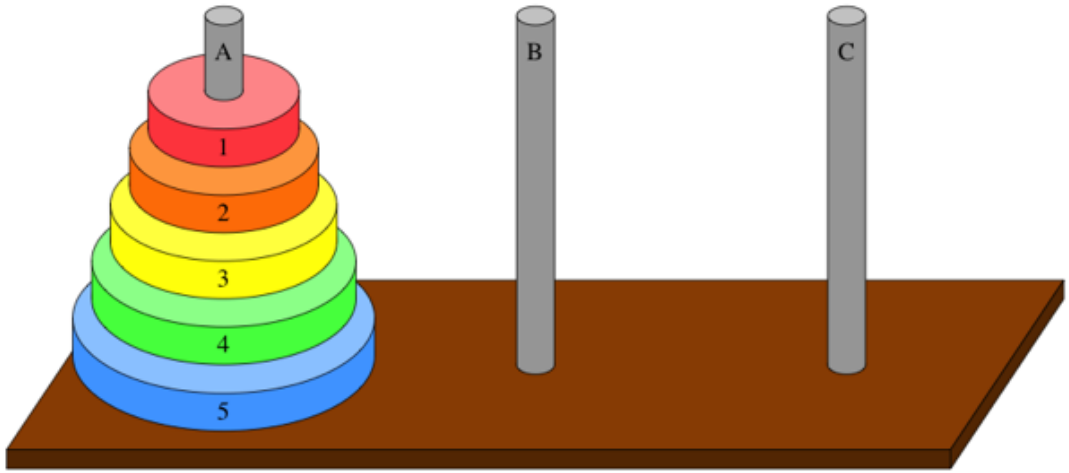


Problema de las torres de Hanoi

Problema de las torres de Hanoi

Tenemos un conjunto de tres torres.

- La primera tiene apiladas n fichas de mayor a menor tamaño.
- El problema consiste en pasar las n fichas de la primera a la tercera torre teniendo en cuenta
  - En cada paso solo se puede mover una ficha.
  - Una ficha de un tamaño no puede apilarse sobre otra de menor tamaño.

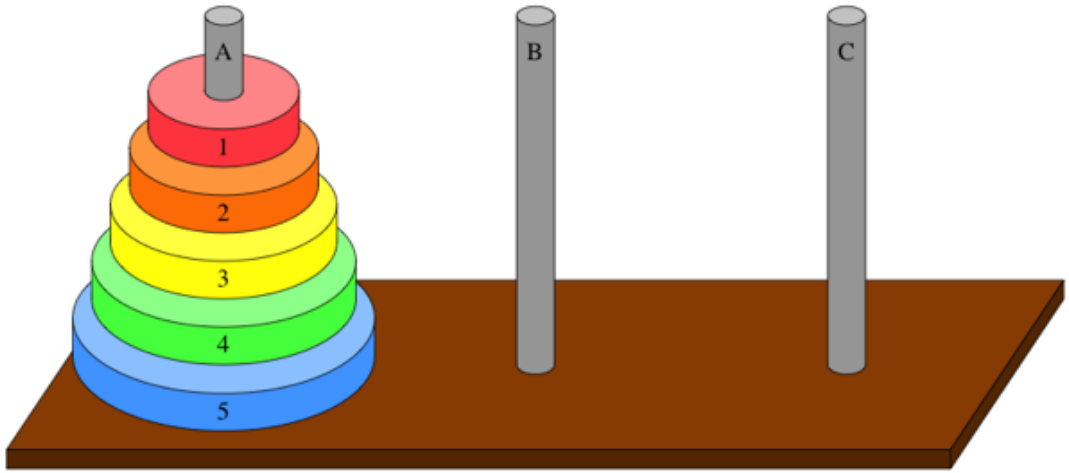


Problema de las torres de Hanoi

Problema de las torres de Hanoi

Tenemos un conjunto de tres torres.

- La primera tiene apiladas n fichas de mayor a menor tamaño.
- El problema consiste en pasar las n fichas de la primera a la tercera torre teniendo en cuenta
  - En cada paso solo se puede mover una ficha.
  - Una ficha de un tamaño no puede apilarse sobre otra de menor tamaño.



```
void hanoi(int m, int inicial, int final){
   if(m==1)
      cout << "Ficha de " << inicial << " a " << final << endl;
   else {
      hanoi(m-1, inicial, 6-inicial-final);
      cout << "Ficha de " << inicial  << " a " << final << endl;
      hanoi(m-1, 6-inicial-final, final);
   }
}
int main(){
    int n;
    cout << "Número de fichas: ";
    cin >> n;
    hanoi(n, 1, 3);
}
```

```
Número de fichas: 3
Ficha de 1 a 3
Ficha de 1 a 2
Ficha de 3 a 2
Ficha de 1 a 3
Ficha de 2 a 1
Ficha de 2 a 3
Ficha de 1 a 3
```

```
void hanoi(int m, int inicial, int final){
   if(m==1)
      cout << "Ficha de " << inicial << " a " << final << endl;
   else {
      hanoi(m-1, inicial, 6-inicial-final);
      cout << "Ficha de " << inicial  << " a " << final << endl;
      hanoi(m-1, 6-inicial-final, final);
   }
}
int main(){
    int n;
    cout << "Número de fichas: ";
    cin >> n;
    hanoi(n, 1, 3);
}
```

```
Número de fichas: 3
Ficha de 1 a 3
Ficha de 1 a 2
Ficha de 3 a 2
Ficha de 1 a 3
Ficha de 2 a 1
Ficha de 2 a 3
Ficha de 1 a 3
```

Soluciones especialmente malas con recursividad

Soluciones especialmente malas con recursividad

Hay que tener cuidado de que la solución recursiva no resuelva varias veces el mismo subproblema en distintas llamadas recursivas.

Sucesión de Fibonacci

$$F(n) = \begin{cases} 1 & \text{si } n = 0 \text{ o } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases}$$

```
int Fibonacci (int n){
   if(n<2)
      return 1;
   else
      return Fibonacci(n-1)+Fibonacci(n-2);
}
```

Soluciones especialmente malas con recursividad

Soluciones especialmente malas con recursividad

Hay que tener cuidado de que la solución recursiva no resuelva varias veces el mismo subproblema en distintas llamadas recursivas.

Sucesión de Fibonacci

$$F(n) = \begin{cases} 1 & \text{si } n = 0 \text{ o } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases}$$

```
int Fibonacci (int n){
   if(n<2)
      return 1;
   else
      return Fibonacci(n-1)+Fibonacci(n-2);
}
```

Soluciones especialmente malas con recursividad

Soluciones especialmente malas con recursividad

Hay que tener cuidado de que la solución recursiva no resuelva varias veces el mismo subproblema en distintas llamadas recursivas.

Sucesión de Fibonacci

$$F(n) = \begin{cases} 1 & \text{si } n = 0 \text{ o } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases}$$

```
int Fibonacci (int n){
   if(n<2)
      return 1;
   else
      return Fibonacci(n-1)+Fibonacci(n-2);
}
```

Cita sobre recursividad

Cita sobre recursividad (L. Peter Deutsch: creador de Ghostscript)

"La iteración es humana, la recursión, divina"