

CS2013: Programación III

Teoría: Programación Genérica II

José Chávez

UTEC

Contenido

- Sobrecarga de Funciones Plantilla
- Especialización
- *Variadic Templates*
- *Fold Expressions*

Sobrecarga de Funciones Plantilla

Sobrecarga de Funciones

Una función template puede ser sobrecargada por otra función template o por una función no-template.

Sobrecarga de Funciones

Una función template puede ser sobrecargada por otra función template o por una función no-template.

Recuerde que, para sobrecargar una función, esta debe tener el mismo nombre, pero con distinta cantidad/tipo de parámetros.

Ejemplo 1

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}
int Max(char a, int b){
    cout << "T2 Max(T1 a, T2 b): ";
    return a > b ? a: b;
}

int main(){
    cout << Max(2, 1) << endl;
    cout << Max(0.7, 1.5) << endl;
    cout << Max('a', 1) << endl;

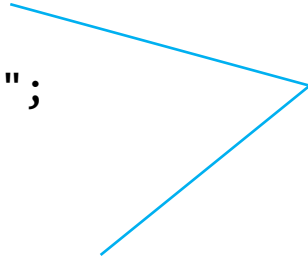
    return 0;
}
```

Ejemplo 1

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}
int Max(char a, int b){
    cout << "T2 Max(T1 a, T2 b): ";
    return a > b ? a: b;
}

int main(){
    cout << Max(2, 1) << endl;
    cout << Max(0.7, 1.5) << endl;
    cout << Max('a', 1) << endl;

    return 0;
}
```



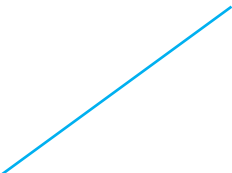
Uno puede declarar varias funciones-plantilla o funciones ordinarias con el mismo nombre.

Ejemplo 1

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}
int Max(char a, int b){
    cout << "T2 Max(T1 a, T2 b): ";
    return a > b ? a: b;
}

int main(){
    cout << Max(2, 1) << endl;
    cout << Max(0.7, 1.5) << endl;
    cout << Max('a', 1) << endl;

    return 0;
}
```



En estos casos la resolución de sobrecarga se realiza de acuerdo a la deducción de argumento de plantilla.

Ejemplo 1

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}
int Max(char a, int b){
    cout << "T2 Max(T1 a, T2 b): ";
    return a > b ? a: b;
}

int main(){
    cout << Max(2, 1) << endl;
    cout << Max(0.7, 1.5) << endl;
    cout << Max('a', 1) << endl;

    return 0;
}
```

Comparar dos enteros

Ejemplo 1

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}
int Max(char a, int b){
    cout << "T2 Max(T1 a, T2 b): ";
    return a > b ? a: b;
}

int main(){
    cout << Max(2, 1) << endl;
    cout << Max(0.7, 1.5) << endl;
    cout << Max('a', 1) << endl;

    return 0;
}
```


Comparar dos
numeros reales

Ejemplo 1

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}
int Max(char a, int b){
    cout << "T2 Max(T1 a, T2 b): ";
    return a > b ? a: b;
}
```

```
int main(){
    cout << Max(2, 1) << endl;
    cout << Max(0.7, 1.5) << endl;
    cout << Max('a', 1) << endl;

    return 0;
}
```



El compilador encontrará la
función que se ajuste mejor a los
parámetros

Ejemplo 1

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}
int Max(char a, int b){
    cout << "T2 Max(T1 a, T2 b): ";
    return a > b ? a: b;
}

int main(){
    cout << Max(2, 1) << endl;
    cout << Max(0.7, 1.5) << endl;
    cout << Max('a', 1) << endl;

    return 0;
}
```

Comparar un carácter y un entero

Especialización

Especialización en funciones

Cuando existen varias sobrecargas a una función que hacen un match con una llamada, el compilador escogerá a la más *especializada*.

Ejemplo 2

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}

template<>
char Max(char a, char b){
    cout << "Especialización Max(char a, char b): ";
    return a > b ? a: b;
}

int main(){
    cout << Max(2, 1) << endl;
    cout << Max('a', 'b') << endl;
    cout << Max(0.7, 1.5) << endl;

    return 0;
}
```

Ejemplo 2

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}

template<>
char Max(char a, char b){
    cout << "Especialización Max(char a, char b): ";
    return a > b ? a: b;
}

int main(){
    cout << Max(2, 1) << endl;
    cout << Max('a', 'b') << endl;
    cout << Max(0.7, 1.5) << endl;

    return 0;
}
```

A diferencia de la sobrecarga, debe existir un emparejamiento entre la función template y la especialización

Ejemplo 2

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}
```

Al implementar la especialización, el compilador ya no necesita crearla

```
template<>
char Max(char a, char b){
    cout << "Especialización Max(char a, char b): ";
    return a > b ? a: b;
}
```

```
int main(){
    cout << Max(2, 1) << endl;
    cout << Max('a', 'b') << endl;
    cout << Max(0.7, 1.5) << endl;

    return 0;
}
```

Ejemplo 2

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}
```

Recuerde que una **especialización** es una **instancia** de una plantilla

```
template<>
char Max(char a, char b){
    cout << "Especialización Max(char a, char b): ";
    return a > b ? a: b;
}
```

```
int main(){
    cout << Max(2, 1) << endl;
    cout << Max('a', 'b') << endl;
    cout << Max(0.7, 1.5) << endl;

    return 0;
}
```

Ejemplo 2

```
template<class T>
T Max(T a, T b){
    cout << "T Max(T a, T b): ";
    return a > b ? a: b;
}
```

```
template<>
char Max(char a, char b){
    cout << "Especialización Max(char a, char b): ";
    return a > b ? a: b;
}
```

```
int main(){
    cout << Max(2, 1) << endl;
    cout << Max('a', 'b') << endl;
    cout << Max(0.7, 1.5) << endl;

    return 0;
}
```

Ambas funciones son viables, sin embargo, la segunda está más **especializada**

Especialización en Clases

Del mismo modo que existe especialización en función, existen especialización en clases.

Ejemplo 3

```
template<class T>
struct Foo{
    Foo(){ cout << "Plantilla Original" << endl;}
};

template<>
struct Foo<int>{
    Foo(){ cout << "Plantilla Especializada" << endl;}
};

int main(){
    Foo<float> obj1;
    Foo<char> obj2;
    Foo<int> obj3;
    return 0;
}
```

Ejemplo 3

```
template<class T>
struct Foo{
    Foo(){ cout << "Plantilla Original" << endl;}
};

template<>
struct Foo<int>{
    Foo(){ cout << "Plantilla Especializada" << endl;}
};

int main(){
    Foo<float> obj1;
    Foo<char> obj2;
    Foo<int> obj3;
    return 0;
}
```

Salida:

"Plantilla Original"

Ejemplo 3

```
template<class T>
struct Foo{
    Foo(){ cout << "Plantilla Original" << endl;}
};

template<>
struct Foo<int>{
    Foo(){ cout << "Plantilla Especializada" << endl;}
};

int main(){
    Foo<float> obj1;
    Foo<char> obj2;
    Foo<int> obj3;
    return 0;
}
```

Salida:

"Plantilla Original"

Ejemplo 3

```
template<class T>
struct Foo{
    Foo(){ cout << "Plantilla Original" << endl;}
};

template<>
struct Foo<int>{
    Foo(){ cout << "Plantilla Especializada" << endl;}
};

int main(){
    Foo<float> obj1;
    Foo<char> obj2;
    Foo<int> obj3;
    return 0;
}
```

Salida:

"Plantilla Especializada"

Ejemplo 3

```
template<class T>
struct Foo{
    Foo(){ cout << "Plantilla Original" << endl;}
};

template<>
struct Foo<int>{
    Foo(){ cout << "Plantilla Especializada" << endl;}
};

int main(){
    Foo<float> obj1;
    Foo<char> obj2;
    Foo<int> obj3;
    return 0;
}
```

El compilador realiza una copia de la clase cada vez que el tipo de dato es distinto (float, char)

Ejemplo 3

```
template<class T>
struct Foo{
    Foo(){ cout << "Plantilla Original" << endl;}
};

template<>
struct Foo<int>{
    Foo(){ cout << "Plantilla Especializada" << endl;}
};

int main(){
    Foo<float> obj1;
    Foo<char> obj2;
    Foo<int> obj3;
    return 0;
}
```

En caso exista una especialización, el compilador **primero** verifica si existe un match con este tipo de dato

Variadic Templates

Pack de Parámetros

```
void foo();  
void foo(int x1);  
void foo(int x1, float x2);  
void foo(int x1, float x2, double x3, char x4, string x5, int x6);
```

Pack de Parámetros

```
void foo();  
void foo(int x1);  
void foo(int x1, float x2);  
void foo(int x1, float x2, double x3, char x4, string x5, int x6);
```



```
template<typename... T>  
void foo(Ts... xs);
```

Pack de Parámetros

```
void foo();  
void foo(int x1);  
void foo(int x1, float x2);  
void foo(int x1, float x2, double x3, char x4, string x5, int x6);
```



```
template<typename... T>  
void foo(Ts... xs);
```

Un *variadic template* es una función plantilla o clase plantilla que acepta un pack de parámetros

Pack de Parámetros

```
void foo();  
void foo(int x1);  
void foo(int x1, float x2);  
void foo(int x1, float x2, double x3, char x4, string x5, int x6);
```



```
template<typename... T>  
void foo(Ts... xs){  
    cout << sizeof...(xs);  
}
```

Podemos obtener la cantidad de
parámetros ingresados

Pack de Parámetros

```
void foo();  
void foo(int x1);  
void foo(int x1, float x2);  
void foo(int x1, float x2, double x3, char x4, string x5, int x6);
```



```
template<typemane... T>  
void foo(Ts... xs){  
    ...  
}
```

¿Cómo iteramos a través de los parámetros?

Pack de Parámetros

```
void foo();  
void foo(int x1);  
void foo(int x1, float x2);  
void foo(int x1, float x2, double x3, char x4, string x5, int x6);
```



```
template<typemane... T>  
void foo(Ts... xs){  
    ...  
}
```

No es posible de manera directa. Para eso están los *Fold Expressions*

Fold Expressions

Fold Expressions

Reduce/resume un pack de parámetros a través de operadores binarios.

Fold Expressions

Reduce/resume un pack de parámetros a través de operadores binarios.

Algunos operadores binarios (dos operandos):

`+, -, *, /, %, &, |, =, <, >, +=, ==, !=, &&, ...`

Sintaxis

- (pack **op** ...)
- (... **op** pack)
- (pack **op** ... **op** init)
- (init **op** ... **op** pack)

Sintaxis

El operador
binario

- (pack **op** ...)
- (... **op** pack)
- (pack **op** ... **op** init)
- (init **op** ... **op** pack)

Sintaxis

- (pack op ...)
- (... op pack)
- (pack op ... op init)
- (init op ... op pack)

El pack de
parámetros

Sintaxis

- (pack **op** ...)
- (... **op** pack)
- (pack **op** ... **op** init)
- (init **op** ... **op** pack)

Todo lo que no sea
un **pack** ni un **op**

Expansión

(pack op ...)



$(a_1 \text{ op } (\dots \text{ op } (a_{n-1} \text{ op } a_n)))$

Expansión

$(\dots \text{op pack})$



$(((a_1 \text{ op } a_2) \text{ op } \dots) \text{ op } a_n)$

Expansión

(pack op ... op *init*)



(a_1 op (... op (a_{n-1} op (a_n op *init*))))

Expansión

(*init* *op* ... *op* pack)



((((*init* *op* a_1) *op* a_2) *op* ...) *op* a_n)

Ejemplo 4

```
template<class... Ts>
auto suma(Ts... args){
    return (args+...);
}

int main(){
    cout << "Utilizando 'suma'" << endl;
    cout << suma(1) << endl;
    cout << suma(1,2) << endl;
    cout << suma(1,2,3.5,0.5,5) << endl;

    return 0;
}
```

Ejemplo 4

```
template<class... Ts>
auto suma(Ts... args){
    return (args+...);
}
```

```
int main(){
    cout << "Utilizando 'suma'" << endl;
    cout << suma(1) << endl;
    cout << suma(1,2) << endl;
    cout << suma(1,2,3.5,0.5,5) << endl;

    return 0;
}
```

¿A que tipo de f.e. pertenece?

- a) (pack op ...)
- b) (... op pack)
- c) (pack op ... op init)
- d) (init op ... op pack)

Ejemplo 4

```
template<class... Ts>
auto suma(Ts... args){
    return (args+...);
}

int main(){
    cout << "Utilizando 'suma'" << endl;
    cout << suma(1) << endl;
    cout << suma(1,2) << endl;
    cout << suma(1,2,3.5,0.5,5) << endl;

    return 0;
}
```

Resultado:

1

Ejemplo 4

```
template<class... Ts>
auto suma(Ts... args){
    return (args+...);
}

int main(){
    cout << "Utilizando 'suma'" << endl;
    cout << suma(1) << endl;
    cout << suma(1,2) << endl;
    cout << suma(1,2,3.5,0.5,5) << endl;

    return 0;
}
```

Resultado:

3

Ejemplo 4

```
template<class... Ts>
auto suma(Ts... args){
    return (args+...);
}

int main(){
    cout << "Utilizando 'suma'" << endl;
    cout << suma(1) << endl;
    cout << suma(1,2) << endl;
    cout << suma(1,2,3.5,0.5,5) << endl;

    return 0;
}
```

Resultado:

12

Ejemplo 5

```
template<typename... Ts>
void print(Ts... args){
    (cout << ... << args) << endl;
}
```

```
int main(){
    print();
    print(1,2,3,4,5);
    print("Hola", 1, "Mundo");

    return 0;
}
```

Ejemplo 5

```
template<typename... Ts>
void print(Ts... args){
    (cout << ... << args) << endl;
}
```

```
int main(){
    print();
    print(1,2,3,4,5);
    print("Hola", 1, "Mundo");

    return 0;
}
```

¿A que tipo de f.e. pertenece?

- a) (pack op ...)
- b) (... op pack)
- c) (pack op ... op init)
- d) (init op ... op pack)

Ejemplo 5

```
template<typename... Ts>
void print(Ts... args){
    (cout << ... << args
```

Resultado:

```
}

int main(){
    print();
    print(1,2,3,4,5);
    print("Hola", 1, "Mundo");

    return 0;
}
```

Ejemplo 5

```
template<typename... Ts>
void print(Ts... args){
    (cout << ... << args) << endl;
}

int main(){
    print();
    print(1,2,3,4,5);
    print("Hola", 1, "Mundo");

    return 0;
}
```

Resultado:

12345

Ejemplo 5

```
template<typename... Ts>
void print(Ts... args){
    (cout << ... << args) << endl;
}

int main(){
    print();
    print(1,2,3,4,5);
    print("Hola", 1, "Mundo");

    return 0;
}
```

Resultado:

Hola1Mundo

Ejemplo 6 (De forma recursiva)

```
void print(){cout << endl;}

template<typename T, typename... Ts>
void print(T arg, Ts... args){
    cout << arg;
    print(args...);
}

int main(){
    print();
    print(1,2,3,4,5);
    print("Hola", 1, "Mundo");

    return 0;
}
```

Ejemplo 6 (De forma recursiva)

```
void print(){cout << endl;}

template<typename T, typename... Ts>
void print(T arg, Ts... args){
    cout << arg;
    print(args...);
}
```

Cuando el pack no contenga ningún parámetro, se llamará a la función `print()`

```
int main(){
    print();
    print(1,2,3,4,5);
    print("Hola", 1, "Mundo");

    return 0;
}
```