Uso de templates

Temas:

Templates

Ejemplos con funciones simples

Desarrollo de funciones de vectores con templates

¿Qué son los templates?

Uno de los problemas clásicos con que nos enfrentamos hasta ahora es el desarrollo de funciones idénticas para diferentes tipos de datos. Por ejemplo si desarrollamos la función suma que reciba 2 valores enteros, y devuelva la suma (también un valor entero), y queremos utilizarla para sumar valores en punto flotante, no lo podemos hacer.

Para ser más claros: se deben desarrollar tantas funciones SUMA como tipos de datos tengamos, así tendremos fSumaEnteros, fSumaFloat, fSumaDouble,….etc.

Y cuando trabajamos con estructuras complejas, por ejemplo vectores pasa exactamente lo mismo, deberemos escribir tantos buscar como tipos de datos haya, y acá si que hay tantos como algoritmos debamos desarrollar.

Algunos pensaran que el tema no requiere de tanto análisis, si en definitiva se trata de "copiar y pegar". Y listo, no se debería hablar más de esto. Pero, y siempre hay un pero (por suerte), qué pasa con la reutilización, para qué hablamos tanto de la programación modular y las ventajas de escribir funciones genéricas y luego reutilizarlas.

Por suerte en C++ (en C esto no existía) se pueden desarrollar verdaderas funciones genéricas a través del uso de plantillas o templates (en inglés) que es como se reconoce en programación. Así pues podemos escribir una única función suma de 2 valores e indicar en la invocación el tipo de datos que se le envía.

O bien podemos escribir las funciones para el uso de vectores una única vez y al ser invocadas indicar el tipo de datos del vector. Y hasta podremos escribir una única función ordenar (por ejemplo inserción) y que "sirva" para ordenar por más de un campo.

En síntesis, mediante el uso de template podremos armar una librería de funciones que permita utilizar las mismas en diferentes contextos de datos.

¿Qué son los punteros a funciones?

Las funciones pueden ser pasadas como parámetros a otras funciones para que éstas las invoquen.

Se pueden crear apuntadores a funciones, en lugar de direccionar datos, los punteros a funciones apuntan a código ejecutable. Un puntero a una función es un apuntador cuyo valor es el nombre de la función.

Los apuntadores permiten pasar una función como argumento a otra función.

Mediante esta característica que ofrece el C++ podremos desarrollar funciones más genéricas.

Por ejemplo desarrollaremos una única función buscar en un vector de registros que se pueda utilizar para buscar por diferentes campos.

Trataremos de manera similar el ordenamiento, de forma tal que podamos ordenar por uno o varios campos sin necesidad de desarrollar varias funciones de ordenamiento para cada método.

Mediante el uso de punteros a funciones podremos armar una librería de funciones que permita utilizar las mismas en diferentes contextos de datos y funcionalidad.

Vamos a trabajar con ejemplos:

Primer ejemplo, operaciones aritméticas básicas, comparaciones e intercambio:

¿ Cómo resolver una función que permita utilizarla con parámetros y variables enteras y en punto flotante?

- Desarrollo de las funciones fSuma, Resta, fProducto y fDivision (similares a suma).

- Desarrollo de la función "fOperacion" que "permite" realizar cualquiera de las operaciones básicas.

- Desarrollo de funciones fMax, fMin, fCambio para comparar e intercambiar dos valores.

Segundo ejemplo, operaciones con vectores:

-¿Cómo redefinir las funciones básicas de vectores mediante templates?

- Desarrollo de la función búsqueda para diferente tipo de datos

- Desarrollo de la función ordenar para uno o varios campos.

Prestemos mucha atención al uso de templates, al pasaje de "funciones" a las funciones a desarrollar y a la forma en que se invoca. Esta manera de desarrollo permite reutilizar el código.

Primer ejemplo, operaciones aritméticas básicas, comparaciones e intercambio:

¿ Cómo resolver una función que permita utilizarla con parámetros y variables enteras y en punto flotante?

Desarrollar funciones para las cuatro operaciones básicas (suma, resta, multiplicación y división) que se puedan utilizar con tipos de datos entero y/o punto flotante, sin necesidad de "reescribir" cada función.

- funciones para las operaciones básicas: fSuma, Resta, fProducto y fDivision.

- función "fOperacion" que "permite" realizar cualquiera de las cuatro operaciones básicas.

- función "genérica" fOperacion que permite resolver cualquiera de las cuatro operaciones básicas.

¿Cómo desarrollar funciones para comparar e intercambiar dos valores, que permita utilizarla con parámetros y variables de diverso tipo: entero, flotante, cadena de caracteres?

- función que determina máximo entre dos valores: fMax

- función que determina mínimo entre dos valores: fMin

- función que intercambia dos valores: fCambio

Desarrollamos una función que recibe dos valores y devuelve la suma

```
int fSuma(int a, int b) {
   int res = a + b;
   return res ;

} ;
```

¡¡Parece fácil, y realmente lo es!!

Vamos a probar cómo funciona

```
int main () {
  int a = 10 ;
  int b = 15
  int  res = fSuma(a, b) ;
 cout<<res<<endl;
 return 0;
} ;
```

¡¡Perfecto!!

Volvamos a probar cómo funciona

```
int main () {
  float a = 10.0;
  float b = 15.0;
  float  res = fSuma(a, b) ;
 cout<<res<<endl;
 return 0;
} ;
```

¡¡Ups!!, no compila

Prestemos atención:

En el primer caso usamos variables enteras y todo bien.

Pero en el segundo caso las variables son float.

Con templates: el tipo de la función, de los parámetros y de la variable local son "genéricos

```
template <typename TS>
TS fSuma(TS a, TS b) {
   TS res = a + b;
   return res ;
} ;
```

Con números enteros

```
int main () {
  int a = 10 ;
  int b = 15
  int  res = fSuma<int>(a, b) ;
 cout<<b<<endl;
 return 0;
} ;
```

¡¡Perfecto!!

Con números reales

```
int main () {
  float a = 10.0;
  float b = 15.0;
  float  res = fSuma<float>(a, b) ;
 cout<<res<<endl;
 return 0;
} ;
```

¡¡Perfecto!!

Prestemos atención:

En este caso se le indica a la función que el tipo es "genérico", y se define cuando se invoca.

Definamos una función que reciba 2 variable y devuelva la resta

```
template <typename TS>
TS fResta(TS a, TS b) {
   TS res = a - b;
   return res ;
} ;
```

Definamos una función que reciba 2 variable y devuelva el producto

```
template <typename TS>
TS fProducto(TS a, TS b) {
   TS res = a * b;
   return res ;
} ;
```

Definamos una función que reciba 2 variable y devuelva la división

```
template <typename TS>
TS fDivision(TS a, TS b) {
   TS res ;
  if (b!=0) res = a/- b;
  else res = 9999999.99;
   return res ;
} ;
```

No es ni más ni menos que aplicar la misma idea que usamos para la suma

---

Mediante la función fOperacion vamos a poder realizar cualquiera de las cuatro operaciones básicas, solamente mencionando cuál se quiere realizar

```
template <typename TX>
TX fOperacion(TX a, TX b, TX (*hacer) (TX,TX))   {
  TX res = hacer(a, b) ;
  return res ;
} ;
```

Observemos la línea de parámetros: aparece (*hacer)(TX, TX),

Donde:
- el símbolo * representa un "puntero" a una función;
- "hacer" representa una función;
- entre paréntesis los tipos de los parámetros

Y lo podemos usar así:

```
int main () {
    cout<<fOperacion<int>(10,2,fSuma)<<endl;
   cout<<fOperacion<int>(10,2,fResta)<<endl;
   cout<<fOperacion<float>(10.0,2.0,fSuma)<<endl;
   cout<<fOperacion<float>(10.2,2.0,fResta)<<endl;
 return 0;
} ;
```

Veamos algunas funciones más:
- mayor entre dos valores
- menor entre dos valores
- intercambio de dos valores

Devuelve el máximo de dos valores

```
template <typename T>
T fMax(T x, T y) {
 if (x < y)
   return y ;
 else return x;
}
```

Devuelve el mínimo de dos valores

```
template <typename T>
T fMin(T x, T y) {
 if (x > y)
   return y ;
 else return x;
}
```

Intercambia dos valores

```
template <typename T>
void fCambio(T & x, T & y) {
 T aux;
 aux = x ;
 x   = y ;
 y   = aux;
}
```

Y lo podemos usar así:

```
int main () {
  int a = 10;
  int b = 15;
  setlocale(LC_CTYPE, "Spanish");
  cout<<"Intercambios, máximos, mínimos"<<endl;
  cout<<"Máximo "<<fMax<int>(a, b)<<endl;
  cout<<"Nínimo "<<fMin<int>(a, b)<<endl;
  cout<<"Antes del cambio ==> a="<<a<<" b="<<b<<endl;
  fCambio<int>(a, b) ;
  cout<<"Después del cambio ==> a="<<a<<" b="<<b<<endl;
  return 0;
}
```

```
Intercambios, máximos, mínimos
Máximo 15
Nínimo 10
Antes del cambio ==> a=10 b=15
Después del cambio ==> a=15 b=10

--------------------------------
Process exited after 4.92 seconds with return value 0
Presione una tecla para continuar . . .
```

```
int main () {
  setlocale(LC_CTYPE, "Spanish");
  string a = "Julieta";
  string b = "Facundo";
  cout<<"Intercambios, máximos, mínimos"<<endl;
  cout<<"Máximo "<<fMax<string>(a, b)<<endl;
  cout<<"Nínimo "<<fMin<string>(a, b)<<endl;
  cout<<"Antes del cambio ==> a="<<a<<" b="<<b<<endl;
  fCambio<string>(a, b) ;
  cout<<"Después del cambio ==> a="<<a<<" b="<<b<<endl;
  return 0;
}
```

```
Intercambios, máximos, mínimos
Máximo Julieta
Nínimo Facundo
Antes del cambio ==> a=Julieta b=Facundo
Después del cambio ==> a=Facundo b=Julieta

--------------------------------
Process exited after 18.28 seconds with return value 0
Presione una tecla para continuar . . .
```

Segundo ejemplo, operaciones con vectores

¿Cómo redefinir las funciones básicas de vectores mediante templates?

- Desarrollo de la función búsqueda para diferente tipo de datos

- Desarrollo de la función ordenar para uno o varios campos.

- Desarrollo de la función mostrar que recorre el vector desde la posición "desde" hasta la posición "hasta" y muestra los datos de la celda, para diversos tipos de datos

Es muy importante detenerse en el programa principal (main) y observar como se utiliza siempre la misma función independientemente del tipo de datos de los parámetros, y como se pasa a la función que muestra los datos de un registro, "punteros" a otras funciones (que conocen los tipos de datos y "saben" cómo listarlos).

Función "mostrar" y diversas funciones para mostrar:

Función "Mostrar", recorre un vector "genérico y por cada celda "invoca" a la función "mostrarFila".

En la lista de parámetros se envía a la función mostrar el vector "desde" y "hasta" qué posición recorrer y un "puntero" a la función "mostrarFila"

Función "mostrar", recorre el vector e invoca a la función "mostrarFila

```
template <typename T, typename R>
void mostrar(T vec, int desde, int len, void (*mostrarFila)(R)){
  for(int i=desde; i<len; i++){
     mostrarFila(vec[i]);
  }
 }
```

Diversos tipos de datos para trabajar con template

```
Registro de estudiantes
struct tREstudiante {
    int legajo;
    string nombre;
    long dni;
    tFecha   fNac;
};

typedef  tREstudiante tVEstudiante[2];

typedef  int tVEdades[5];
```

Función mostrarEstudiante

```
void mostrarEstudiante(tREstudiante r){
   cout<<r.legajo<<" "<< r.nombre<<" "<< r.dni<<endl ;
 }
```

Función mostrarEstudianteNac

```
void mostrarEstudianteNac(tREstudiante r){
 cout<<r.legajo<<" "<< r.nombre<<" "<<
                r.fNac.dia<<" "<< r.fNac.mes<<" "<<
                r.fNac.anio<<endl ;
 }
```

Función "mostrarEdad", muestra el valor recibido.

```
void mostrarEdad(int r){
    printf("%5d \n", r) ;
 }
```

Se pueden definir diferentes tipos de datos y mediante la función mostrar, recorrer los mismos.

Se pueden definir diversas funciones para diversos tipos de datos y mediante mostrar ejecutarlas.

Programa que muestra estudiante de dos formas diferentes, y edades.

```
int main () {
 tVEstudiante vEstudiante= {{8, "Ana María", 23235426, { 20 ,3  ,2020  }},
                {12, "Carlos Antonio", 33232326, { 18 ,6 , 2018  }}};
 setlocale(LC_CTYPE, "Spanish");
 cout<<"Mostrar estudiantes (legajo, nombre, dni)"<<endl;
 mostrar<tVEstudiante, tREstudiante>(vEstudiante, 0, 2,  mostrarEstudiante);

 cout<<endl<<"Mostrar estudiantes (legajo, nombre, fecha nacimiento)"<<endl;
 mostrar<tVEstudiante, tREstudiante>(vEstudiante, 0, 2,  mostrarEstudianteNac);

 tVEdades vEdad = {23, 32, 44, 15, 19} ;
 cout<<endl<<"Mostrar edades"<<endl;
 mostrar<tVEdades,  int>(vEdad, 0, 5,  mostrarEdad);
 return 0;
}
```

```
Mostrar estudiantes (legajo, nombre, dni)
8 Ana María 23235426
12 Carlos Antonio 33232326

Mostrar estudiantes (legajo, nombre, fecha nacimiento)
8 Ana María 20 3 2020
12 Carlos Antonio 18 6 2018

Mostrar edades
   23
   32
   44
   15
   19

--------------------------------
Process exited after 3.435 seconds with return value 0
Presione una tecla para continuar . . .
```

Podemos observar en el main del ejemplo que se definen dos tipos de datos diferentes y se les asigna valores al declarar las variables (vEstudiante y vEdad). Se muestran los estudiantes de dos formas diferentes: mediante la función "mostrarEstudiante" y mediante la función "mostrarEstudianteNac".
Se muestran las edades mediante la función "mostrarEdad".
En todos los casos se utiliza la función "mostrar" que recibe diferentes tipos de datos y punteros a diferentes funciones.

Las funciones "puntero" conocen los tipos de datos y "saben" cómo listarlos

Función "buscar" y diversas funciones para buscar:

La función "buscar" recorre el vector recibido entre dos límites (desde y hasta) y busca el valor a buscar. Permite diversos tipos de datos a través de armar "criterios" mediante puntero a las funciones donde están desarrollados los criterios de búsqueda

Función "buscar", recorre el vector e invoca a la función "criterio". Notar que tiene 3 tipos de datos genéricos diferentes: tipo de vector (T), tipo de la celda del vector ®, y tipo de la clave (K)

```
template <typename T, typename R, typename K>
int buscar(T vec, int desde, int len, K v, int (*criterio)(R,K)) {
  int i=desde;
  while( i<len && criterio(vec[i],v)!=0 ){
    i++;
  }
  if (i==len) i=-1;
  return i;
}
```

Diversos tipos de datos para trabajar con template

```
Registro de estudiantes
struct tREstudiante {
    int legajo;
    string carrera;
    string nombre;
    long dni;
    tFecha   fNac;
};
typedef  tREstudiante tVEstudiante[2];

typedef  int tVEdades[5];
```

Función "criterio Nombre", busca por nombre

```
template <typename T>
int criterioNombre(T a1, T a2) {
   int r=0;
   if (a1.nombre<a2.nombre) r = -1 ;
   else if (a1.nombre>a2.nombre) r = 1 ;
   return r ;
}
```

Función "criterioEdad", busca por edad.

```
template <typename T>
int criterioEdad(T a1, T a2) {
   int r=0;
   if (a1<a2) r = -1 ;
   else if (a1>a2) r = 1 ;
   return r ;
}
```

Función "criterioCarreraLegajo", busca por esos campos.

```
template <typename T>
int criterioCarreraLegajo(T a1, T a2) {
  int r=0;
   if ((a1.carrera>a2.carrera) ||  ((a1.carrera==a2.carrera) &&
(a1.legajo>a2.legajo))) r = 1 ;
   else if  ((a1.carrera<a2.carrera) ||  ((a1.carrera==a2.carrera) &&
(a1.legajo<a2.legajo))) r = -1 ;
   return r ;
}
```

Programa que crea una lista de alumnos y los busca por legajo y luego busca y muestra todos los estudiantes de la carrera "Sistemas".:

```
int main() {
    tVAlumno vAlu;
    int len=0;
    cargarVectorAlumnos(vAlu, len) ;
    int i = buscar<tVAlumno, tRAlumno, int>(vAlu, 0, len, 33, criterioBusqLegajo);
    cout << "Buscar un alumno por legajo=33 ("<<i<<")" << endl;
    if (i>-1)
        mostrar<tVAlumno, tRAlumno>(vAlu, i, i+1, mostrarAlumnoCarrera);
    else cout<<"No esta"<<endl;

    i = buscar<tVAlumno, tRAlumno, int>(vAlu, 0, len, 34, criterioBusqLegajo);
    cout << "Buscar un alumno por legajo=34 ("<<i<<")" << endl;
    if (i>-1)
        mostrar<tVAlumno, tRAlumno>(vAlu, i, i+1, mostrarAlumnoCarrera);
    else cout<<"No esta"<<endl;

   cout << "Buscar los alumno de sistemas"<< endl;
   i = buscar<tVAlumno, tRAlumno, string>(vAlu, 0, len, "Sistemas", criterioBusqCarrera);
    while (i!=-1) {
      mostrar<tVAlumno, tRAlumno>(vAlu, i, i+1, mostrarAlumnoCarrera);
      i = buscar<tVAlumno, tRAlumno, string>(vAlu, i+1, len, "Sistemas", criterioBusqCarrera);
    }
}
```

```
==========================================
Buscar un alumno por legajo=33 (1)
==========================================
Industrial, Jose, 33, 92434546
==========================================
Buscar un alumno por legajo=34 (-1)
==========================================
No esta
==========================================
Buscar los alumno de sistemas
==========================================
Sistemas, Angel, 230, 33958074
Sistemas, Carlos, 933, 54457675
Sistemas, Jimena, 532, 22366575
Sistemas, Juan, 134, 45656765
Sistemas, Mariano, 73, 90326568
Presione una tecla para continuar . . .
```

En el ejemplo podemos observar que se realizan dos búsquedas por legajos y en caso de encontrar los legajos buscados se muestran los datos mediante las funciones buscar de las filminas anteriores.

En el último bloque se busca el "primer" alumno de Sistemas, se muestra y se sigue buscando hasta el final del vector.

Funciones para ordenar entre dos valores. Se utilizan la función "ordenar":

La función ordenar es un clásico ordenamiento búrbuja. En este caso recibe en forma genérica los parámetros de ordenamiento y los punteros a las funciones criterios mediante los cuales analiza el orden e intercambia si corresponde.

Criterio para comparar 2 legajos. Se pasa el tipo "genérico" para independizarlo de la función ordenar.

```
template <typename T>
int criterioLegajo(T a1, T a2) {
  int r=0;
  if (a1.legajo<a2.legajo) r = -1 ;
  else if (a1.legajo>a2.legajo) r = 1 ;
  return r ;
}
```

Criterio para comparar 2 nombres. Se pasa el tipo "genérico" para independizarlo de la función ordenar.

```
template <typename T>
int criterioNombre(T a1, T a2) {
  int r=0;
  if (a1.nombre<a2.nombre) r = -1 ;
  else if (a1.nombre>a2.nombre) r = 1 ;
  return r ;
}
```

Criterio para comparar 2 carreras. Se pasa el tipo "genérico" para independizarlo de la función ordenar.

```
template <typename T>
int criterioCarrera(T a1, T a2) {
  int r=0;
  if (a1.carrera>a2.carrera) r = 1 ;
  else if (a1.carrera<a2.carrera) r = -1 ;
  return r ;
}
```

Criterio para comparar por dos campos: carrera y nombre. Permite ordenar por carrera y dentro de la carrera por nombre

```
template <typename T>
int criterioCarreraNombre(T a1, T a2) {
  int r=0;
  if ((a1.carrera>a2.carrera) ||  ((a1.carrera==a2.carrera) &&
                                   (a1.nombre>a2.nombre))) r = 1 ;
  else if  ((a1.carrera<a2.carrera) ||  ((a1.carrera==a2.carrera) &&
                                   (a1.nombre<a2.nombre))) r = -1 ;
  return r ;
}
```

Las funciones representan diversos criterios para determinar el orden entre dos valores (a1 y a2).

En las de más arriba se compara por un único campo conocido, mientras que en "criterioCarreraNombre" se compara por dos campos.

Es posible desarrollar tantas funciones como sean necesarias y una única función "ordenamiento .

Función "ordenar" con parámetros "genéricos" y "puntero" a la función "criterio":

La función "ordenar" permite ordenar diversos tipos de vectores y mediante diferentes criterios:

En la línea de parámetros se observa que el tipo de datos del vector (vec) es T (referenciado en el template) el criterio de ordenamiento es un "puntero" a la función "criterio", la cual recibe dos parámetros de tipo genérico Q

```
template <typename T, typename Q>
void ordenar(T vec, int len, int (*criterio)(Q,Q)) {
    Q aux;
    bool ordenado=false;
   while(!ordenado){
      ordenado=true;
      for(int i=0; i<len-1; i++){
        if( criterio(vec[i],vec[i+1])==1 ){
          aux = vec[i];
          vec[i] = vec[i+1];
          vec[i+1] = aux;
          ordenado = false;
        }
      }
   }
}
```

Este método recorre el vector tantas veces como sea necesario hasta que esté ordenado. Cada vez que encuentra dos desordenados los intercambia

Se puede implementar el uso de template en cualquiera de los métodos conocidos.

```
int main()   {
                tVAlumno vAlu;
    int len=0;
    cargarVectorAlumnos(vAlu, len) ;

   cout << "Mostrar alumnos ordenados por  nombre" << endl;
   ordenar<tVAlumno, tRAlumno>(vAlu, len, criterioNombre);
   mostrar<tVAlumno, tRAlumno>(vAlu, 0, len, ostrarAlumnoNombre);

   cout << "Mostrar alumnos ordenados por  carrera" << endl;
   ordenar<tVAlumno, tRAlumno>(vAlu, len, criterioCarrera);
   mostrar<tVAlumno, tRAlumno>(vAlu, 0, len, mostrarAlumnoCarrera);

   cout << "Mostrar alumnos ordenados por  carrera y legajo" << endl;
   ordenar<tVAlumno, tRAlumno>(vAlu, len, criterioCarreraLegajo);
   mostrar<tVAlumno, tRAlumno>(vAlu, 0, len, mostrarAlumnoLegajo);
}
```

```
Mostrar alumnos ordenados por  nombre
=====================================
Ana, 234, 33435675, Industrial
Angel, 230, 33958074, Sistemas
Carlos, 933, 54457675, Sistemas
Jimena, 532, 22366575, Sistemas
Jose, 33, 92434546, Industrial
Juan, 134, 45656765, Sistemas
Lara, 23, 34698074, Textil
Mariano, 73, 90326568, Sistemas
Marta, 44, 32568764, Textil
```

```
Mostrar alumnos ordenados por  carrera y legajo
===============================================
33, Jose, 92434546, Industrial
234, Ana, 33435675, Industrial
73, Mariano, 90326568, Sistemas
134, Juan, 45656765, Sistemas
230, Angel, 33958074, Sistemas
532, Jimena, 22366575, Sistemas
933, Carlos, 54457675, Sistemas
23, Lara, 34698074, Textil
44, Marta, 32568764, Textil
```

Llegamos al final de la presentación de uso de template.

Repasemos.

- Utilizar template o plantilla permite que las funciones sean genéricas, de esta manera se puedan utilizar para diversos tipos de datos.
- Mediante funciones "puntero" se pueden pasar en la lista de parámetros referencias a otras funciones, lo que permite la reutilización del código.
- Los ejemplos presentados nos muestran que podemos resolver únicas funciones "ordenar", "buscar" o "recorrer" que funcionan correctamente con diferentes tipos de datos y criterios de búsqueda u ordenamiento.
- Para las estructuras de datos que vayamos a ir trabajando se hace muy útil trabajar con template, de esta forma cada función será genérica.

Para ampliar les propongo leer el material del Dr. Oscar Bruno, director de la cátedra de Algoritmos y Estructura de Datos de la FRBA UTN en
https://droscarbruno.files.wordpress.com/2015/02/vectoresmatrices2015.pdf#page=16&zoom=100,109,470

Desde la página 21 desarrolla algoritmos para el manejo de array (vectores) mediante el uso de templates.

Varios ejemplos de este material fueron inspirados en el material mencionado.