Mangling in C++

Posted by Alejandro G. Carlstein Ramos Mejia on October 20, 2010 October 20, 2010 About Programming / C++

In standard ANSI C, if we wish to have two functions with the same name but different number of parameters or type of parameters (called *overloading*), we can't. It is not allowed.

In the other hand, in C++, we can *overload* functions. Overloading a function means that we can have two or more functions sharing the same function name but with different type or different number of parameters:

```
int print_something(void){
...
}

int print_something(char a){
...
}

int print_something(int a){
...
}

int print_something(char a, int b){
...
}
```

Even do all these functions share the same name, they are different; so, how does your compiler keep track of which is which? By using a method called mangling algorithm, unique names are generated as identifiers for each of these functions.

First, lets say that we have a file with some functions such as:

```
#include <stdio.h>

int foo(double d_number);

int main(int argc, char* argv[]){

 double d_value = 4.5f;

 printf('For value %f, we get the number %d\n', d_value, foo(d_value));

  return 0;
}

int foo(double d_number){
  return (int) d_number;
}
```

This would show the follow:

```
For value 4.500000, we get the number 4
```

If we compile the file with gcc for example: *gcc -c main.c*. This will generate a file name main*.o*
Then we can analyse this file and see how the table is created in C for this particular function.

Type *nm file.o* and you will obtain something like this:

```
0000003f T foo
00000000 T main
 U printf
```

Notice the table only indicate the name of foo but doesn't indicate any parameters types

What would append if we compile the same program using g++ instead?
A table would be build by the C++ compiler (in this case we are talking about g++) that would look like the follows:

```
0000003e T _Z3food
 U __gxx_personality_v0
00000000 T main
 U printf
```

Look the information *_Z3food*, the last character 'd' indicate that the parameter is a double

Lets say we modified main.c to main.cpp and we create another function with the same name but different parameters:

```
#include <stdio.h>

int foo(double d_number);
int foo(char d_character, int i_number);

int main(int argc, char* argv[]){
  double d_value = 4.5f;
  printf('For value %f, we get the number %d\n', d_value, foo(d_value));
  return 0;
}

int foo(double d_number){
  return (int) d_number;
}

int foo(char d_character, int i_number){
  return (int) d_character + i_number;
}
```

If we print the table using *nm main.o* we obtain:

```
0000006e T _Z3fooci
0000003e T _Z3food
 U __gxx_personality_v0
00000000 T main
 U printf
```

If you notice, *_Z3fooci* has the last two character a ‘c’ for char and an ‘i’ for integer while *_Z3food* has the last character ‘d’ for double.

If we try to compile this code having both functions with the same name using gcc: *gcc -g main.cpp,* we would obtain an error in compilation:

```
/tmp/ccU0T0Co.o (.eh_frame+0x12): undefined reference to `__gxx_personality_v0'
collect2: ld returned 1 exit status
```

Using *extern “C”*, we can tell g++ which part of the code we wish to compile as regular C. Let say we have to following code:

```
#include <stdio.h>

int foo(double d_number);

extern 'C'{
  int foo(char d_character, int i_number);
}

int main(int argc, char* argv[]){
  double d_value = 4.5f;
  printf('For value %f, we get the number %d\n', d_value, foo(d_value));
  return 0;
}

int foo(double d_number){
  return (int) d_number;
}

extern 'C'{
  int foo(char d_character, int i_number){
    return (int) d_character + i_number;
  }
}
```

This would compile with *gcc -g main.cpp* without problems. If we execute *g++ -c main.cpp* to create the object main.o and later executed *nm main.o* we obtain:

```
0000003e T _Z3food
 U __gxx_personality_v0
0000006e T foo
00000000 T main
 U printf
```

Notice that *_Z3food* was compiled as C++ while *T foo* indicate that that function was compiled as standard C.