Modern C++ - Functions

Functions, parameter passing

Functions

Functions are fundamental building blocks in procedural programming, they allow us to break large, complex code into smaller more maintainable parts, hiding implementation details from the view of a higher abstraction, reuse earlier implemented actions and apply variability by using parameters.

The C++ language has been designed to make functions easy tom implement and use, and effective to call.

A set of functions (and perhaps other global objects) can be compiled in separate translation unit and can be used as a library.

Wording: the name of a parameter in a function declaration is called *parameter*, while the actual value passed in the call is *argument*. The full list of parameters is called *prototype*.

Function definition and declaration

The return type and the parameter list is part of the type of the function.

```
int f();
int *fip();
int ( *pfi )();
```

declare a function with no parameters returning int, a function with no parameters returning pointer to int, and a pointer to function with no parameters returning int. Functions with no intended return value should be declared with **void** return type. Array return type is not allowed.

For declarations, we can use formal parameter names, but they are only for descriptive purposes:

```
int f(int *x, int *y);    // x and y has no role in declaration
int f(int *, int *);      // equivalent to the above
```

For historical reasons, functions with no parameters can be write in two forms:

```
int f();           // function with no parameter
int g(void);       // function with no parameter, C compatible
```

When we want to express that we do not know the exact parameters of a function, like in the case of *printf* we use the *ellipsis* notation:

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

The meaning of the ellipsis is that *zero or more additional parameters of unknown type*. To write such variadic parameter function is tricky (and usually done by the *va_* macros from <*stdarg.h*> header.

When we know nothing about the parameters, we can also use ellipsis:

```
int f();          // function with no parameter
int g(...);       // function with no parameter, C compatible
```

Function call

When a function *f* is called, then the declaration prototype should match to the call:

- The number of the parameters should be the same.
- Each argument of the call shall have a type such that its value may be assigned to an object of the type of the corresponding parameter of the prototype.

If the number of parameters does not match with the number of arguments of the call, the behavior is *undefined*. If the function definition prototype have ellipsis or the argument in the call are incompatible with the correspondent parameter, the behavior is undefined.

If the called function has a prototype then the arguments are implicitly converted, as if by assignment, to the corresponding parameters. Ellipsis stops the conversions, only the default argument promotions are executed.

```
double fahr2cels(double);
//...
printf("%f\n", fahr2cels(36));  // ok, 36 is converted to double
```

The function call is a *sequence point*, i.e. first the argument expressions are evaluated in undefined order, only then the function body is starting to execute. (The comma between parameters are different from the *comma operator*.)

```
( *t[f1()] ) ( f2(), f3(), f4() );
```

The functions *f1*, *f2*, *f3*, *f4* can be called in any order.

Recursive functions are allowed both directly or indirectly:

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

When this function is called with an argument smaller then one, *infinite recursion* happens and the program behavior is undefined.

Parameter passing by value

Arguments by default are passed *by value*, i.e. they are copied from the argument expression to the parameter as it would be a local automatic variable declared in the beginning of the function.

```
#include <iostream>
void increment(int i)
{
  ++i;
  std::cout << "i in increment() " << i << std::endl;
}
int main()
{
  int i = 0;
  increment(i);
  increment(i);
  std:.cout << "i in main() = " << i << std::endl;
  return 0;
}
```

```
$ g++ -ansi -pedantic -Wall -W par.cpp
$ ./a.out
i in increment() = 1
i in increment() = 1
```

```
i in main() = 0
```

We can simulate the parameter passing *by address* with pointers:

```
#include <iostream>
void increment(int *ip)
{
  ++*ip;
  std::cout << "i in increment() = " << *ip << std::endl;
}
int main()
{
  int i = 0;
  increment(&i);
  increment(&i);
  std::cout << "i in main() = " << i << std::endl;
  return 0;
}
```

```
$ g++ -ansi -pedantic -Wall -W par.cpp
$ ./a.out
i in increment() = 1
i in increment() = 2
i in main() = 2
```

Reference parameters

Parameters can be passed *by reference*. In this case the parameter behavior is like the function would define a local refernece variable which is initialized to the actual argument passed to the function.

```
#include <iostream>
void increment(int &ir)
{
  ++ir;
  std::cout << "i in increment() = " << ir << std::endl;
}
int main()
{
  int i = 0;
  increment(i);
  increment(i);
  std::cout << "i in main() = " << i << std::endl;
  return 0;
}
```

```
$ g++ -ansi -pedantic -Wall -W par.cpp
$ ./a.out
i in increment() = 1
i in increment() = 2
i in main() = 2
```

Reference parameters can be changed inside the function and since the actual memory referred by the parameter is out of the function, the changes will be permanent after returning from the function.

Array parameters

Array parameters are passed as pointers to the first array element.

```
#include <stdio.h>
int f( int *t, int i)
{
  return t[i];
}
int main()
{
  int arr[] = {1, 2, 3, 4};
  printf("%d\n", f(arr,1));
  printf("%d\n", f(&arr[0],1));
  return 0;
}
```

```
$ gcc -ansi -std=c99 -Wall -W a.c
$ ./a.out
2
2
```

Therefore these parameter declarations are equivalent:

```
int f( int *t, int i)   { return t[i]; }
int f( int t[], int i)  { return t[i]; }
int f( int t[4], int i) { return t[i]; }
```

```

// array limits are not checked: this is not reported as error
int f( int t[4], int i)  { return t[6]; }
```

Pointers to function

Type of a pointer to function includes the return type and the parameter list.

```
#include <iostream>
void increment(int &ir)
{
  ++ip;
  std::cout << "i in increment() " << ir << std::endl;
}
int main()
{
  void (*fp)(int &); // pointer to void (int*)
  int i = 0;
  fp = increment;
  (*fp)(i);    // call increment()
    fp (i);    // simplified notation
  std::cout << "i in main() " << i << std::endl;
  return 0;
}
```

```
$ g++ -ansi -pedantic -Wall -W par.cpp
$ ./a.out
i in increment() = 1
i in increment() = 2
i in increment() = 3
i in main() = 3
```

A pointer to function can be:

- point to a function by a compatible type.
- assigned to a similar pointer variable.
- checked against NULL pointer.
- indirected in a function call.

Return type

Return values are converted to the return type of the function:

```
double f(void)
{
  int i;
  // ...
  return i;   // converted to double
}
```