



[For more details, refer to “Concepts of Programming Languages” by *Robert Sebesta*]

1 Subprogram definition

A *subprogram definition* consists of:

- *Subprogram header*: specifies the subprogram *kind*, *name*, and *protocol*. The *subprogram kind* is either *function* or *procedure*. A *procedure* is a *function* without *return type*. The *protocol* consists of *parameter profile* and *return type* if it is a *function* in a typed language. The *parameter profile* specifies the number, order, and types (for typed languages) of its *formal parameters*. C++ and Java use the reserved word *void* to indicate no *return type*.
- *Subprogram body*: specifies a sequence of statement which are executed in order when another subprogram *calls* it. A *subprogram call* is the explicit request to execute its *body*. The *calling subprogram* provides the *called subprogram* with *actual parameters* to be bound to its *formal parameters*. Sometimes, the term *parameters* is used for *formal parameters* while the term *arguments* is used for *actual parameters*. The *calling subprogram* is suspended during the execution of the *called subprogram*. *Control* is passed to the *entry point* (usually the first statement) of the *called subprogram*. *Control* returns to the *caller* when the *called subprogram* execution terminates.

The correspondence between *actual* and *formal* parameters is usually done by position. That is, the i^{th} *actual parameter* is bound to the i^{th} *formal parameter*. That method is called *positional parameters*. Another method provided by some languages is called *keyword parameters* which specifies the name of the *formal parameter* to be bound with the *actual parameter*. The advantage of this method is that parameters can appear in any order so the programmer does not need to remember the order of formal parameters, such as the following *Python subprogram call*:

```
Fun(length=my_length, list=my_array, sum=my_sum)
```

In several languages, *default values* can be associated to *formal parameters* which are used whenever the *subprogram* call does not specify the corresponding *actual parameters*.

C# allows a method to accept a variable number of parameters of the same type with the *params* modifier, where the *caller* sends either array or list of expressions:

```
public void DisplayList(params int[] list)
{foreach (int val in list) Console.WriteLine("Val={0}", val);}
```

The above function can be called by passing a *list* or a variable number of parameters such as:

```
obj.DisplayList(2, 4, 5, x-1, 17);
```

2 Subprogram declaration

A *subprogram declaration* provides the *subprogram header* but does not include its *body*. It is required in languages that do not allow forward references to subprograms. C and C++ require all *subprograms* to be either *defined* or *declared* before they are called and inside the same *translation unit* where they are called.

```
double Fun(int, double); // Declaration (called prototype in C++)
int main()
{
    int a=5; double b=9.4;
    Fun(5, b);
    return 0;
}
double Fun(int x, double y) {return x+y;} // Definition
```

A project with multiple *source (cpp) files* in C++ is compiled by the following steps:

- For each *source file*, independently of other *source files*, the *preprocessor* processes the *source file* by expanding all macros (instructions starting with #), usually by simple text substitution, to produce a *translation unit*. For example, consider the following file:

```
#include "mylib.h"
#define Max(A, B) ((A)>(B)?(A):(B))
double F(int y, int z) {return G()+Max(2*y, z/5);}
```

Assuming that the file `mylib.h` consists of the following:

```
double G();
```

The *cpp* file will be expanded by the *preprocessor* to the following *translation unit*:

```
double G();
double F(int y, int z) {return G()+((2*y)>(z/5)?(2*y):(z/5));}
```

- For each *translation unit*, independently of other *translation units*, the *compiler* compiles the *translation unit* to produce an *object file*. The *compiler* requires that all functions are either *defined* or *declared* before its first call in the same *translation unit*, in order to perform *static type checking* to validate *type compatibility* between *actual* and *formal parameters*.
- The *linker* combines all *object files* into one *executable image* (also called *load module*), which is a machine-readable executable file or library. The addresses of called functions (such as `G()`) are not necessarily known during *compilation* because only their *declarations* may be available. Therefore, the *linker* is responsible for *resolving* all *function calls* by calculating the correct addresses and placing them into the corresponding call statements inside the *executable*. To be able to do so, the *definition* of each function must exist exactly in one *translation unit*.

The above *compilation* and *linking* mechanisms reduce *compilation* time by avoiding *recompilation* of *source files* which are not changed, including *source files* for built-in C++ libraries. Only the changed *source files* of a project need to be *recompiled*.

In C++, there is no restriction on the number of *function declarations*. Each *non-inline function* must be defined **exactly once** across all files. *Classes* and *inline functions* must be defined **at most once** per *translation unit*, such that **at least one definition** exists for each entity across all files, and all *definitions* for the same entity are identical.

Inline functions include all functions modified by the reserved word `inline`, and all class member functions *defined* inside the class definition. *Inline functions* differ from other functions because the compiler tries to replace *calls* to *inline functions* by the code of the *function body* itself, which may be useful for optimization only if the number of statements in the *body* is small.

Therefore, it is safe to include *function declarations*, *class definitions* and *inline function definitions* in *header (.h) files* and include them in several *source (cpp) files*.

The *declaration* of a *variable* is also its *definition* except in few cases. Suppose there is a *global variable* that needs to be accessed from a *source file* other than the one including its *definition*. In that case, it is just *declared* (not *defined*) in the new file using the `extern` modifier before accessing it, because each *variable* must be *defined exactly once* across all files.

Similarly, *static* class data members are considered *declared* but not *defined* if their *declarations* appear inside their class *definitions*. Hence, they must be *defined* outside their class *definition*. This is because classes can be *defined* several times in different *translation units* but *variables* cannot.

3 Parameter passing

Formal parameters are characterized by one of three distinct semantic models:

- **In mode**: *Formal parameters* receive data from the corresponding *actual parameters*. This mode can implemented by one of two models:
 - *Pass-by-value*: The value of *actual parameter* is used to initialize the corresponding *formal parameter* by copying.
 - *Pass-by-readonly-reference*: Provides read-only access path to the *actual parameter*.

```
void Fun(int a, const int& b); // C++ In mode
```

- **Out mode**: *Formal parameters* transmit data to the corresponding *actual parameters*. This mode can implemented by a *pass-by-result* model: No value is transmitted to the *formal parameter*, which acts as local variable whose value is transmitted back to the *actual parameter* by copying just before control is transferred back to the caller.
- **In-out mode**: *Formal parameters* receive data from and transmit data to the corresponding *actual parameters*. This mode can implemented by one of three models:
 - *Pass-by-value-result*: The value of *actual parameter* is used to initialize the corresponding *formal parameter* by copying. Then, the value of *formal parameter* is transmitted back to the *actual parameter* by copying just before control is transferred back to the caller.
 - *Pass-by-reference*: Provides access path to the *actual parameter*.
 - *Pass-by-name*: The *actual parameter* is textually substituted for the corresponding *formal parameter*. It is used at compile-time only by C++ macros and templates.

The following example illustrates the parameter passing modes:

```
void Fun(in int a, out int b, in-out int c)
{
    // Initially : a=7, b=has undefined value, c=9

    a=1; b=2; c=3;
    // Now: x=7 (no change), y=8 (no change)
    // z=9 (no change) if c is passed by value—result
    // z=3 (changed) if c is passed by reference

    a=a; b=b; c=c; // Do something
} // Immediately before function returns :
   // x=7 (no change), y=2 (changed), z=3

void main()
{
    int x=7, y=8, z=9;
    Fun(x, y, z); // The above comments trace this call
}
```

4 Implementing subprogram calls

We examine the implementation of *subprogram* calls focusing on the call and return procedures. Initially, we assume that the called *subprograms* do not contain any inner *blocks*. *Subprograms* with inner *blocks* are considered later on.

Each *subprogram* has the following simplified typical *activation record*:

Return variable
Local variables
Parameter variables
Return address

Consider the following C++ function. The numbers shown on the left are the addresses of each instruction. Note that program instructions are loaded into memory and obtain memory addresses before they are executed. The *activation record* of this function is shown on the right:

```
int factorial(int n)
{ // Position 1
1004   if(n<=1) return 1;
1008   int f=factorial(n-1);
1012   int r=n*f;
1016   return r;
} // Position 2
```

Return variable	int	
Local variable	int	f
Local variable	int	r
Parameter variable	int	n
Return address		

Subprogram calls are implemented in the same way for *recursive* and *non-recursive subprograms*, but the main reason for such implementation is to support *recursive subprograms*.

Consider the following C++ program which calls the above function:

```

    int main()
    {
2004      int v=factorial(3);
2008      return 0;
    }

```

Each call to `factorial()` starts by pushing to the *run-time stack* an *activation record instance* of the *activation record* of the `factorial()` function. The *run-time stack* of a specific program is part of the main memory assigned by the operating system to this program and can be used to allocate its *stack-dynamic variables*. An *activation record instance* (ARI) is a specific instance of the *activation record* with specific allocated variables.

The call to `factorial(3)` in instruction 2004 starts by pushing to the *run-time stack* the following *activation record instance*. So, when execution reaches **Position 1** for the **first** time, the *run-time stack* contains:

Return variable	int		
Local variable	int	f	
Local variable	int	r	
Parameter variable	int	n	3
Return address			2008

The *return address* is the address of the instruction that follows the *function call* instruction. This address will be used by the compiler to know where it should continue execution (pass *control*) after the *function call* terminates.

Since the size of an *activation record instance* for a specific function is known before the *function call* (actually it is known at *compile time*), only one memory allocation is needed to allocate the whole *activation record instance* which is efficient.

Then, control reaches instruction 1008 then calls `factorial(2)` which starts by pushing another *activation record instance* to the *run-time stack*. So, when execution reaches **Position 1** for the **second** time, the *run-time stack* contains:

Return variable	int		
Local variable	int	f	
Local variable	int	r	
Parameter variable	int	n	2
Return address			1012

Return variable	int		
Local variable	int	f	
Local variable	int	r	
Parameter variable	int	n	3
Return address			2008

For the recursion logic to work, each *function call* must have its distinct set of parameters and local variables. However, while executing a specific *function call*, its associated set of variables always exist at the *activation record instance* at the **top** of the *stack*. Therefore, the compiler can use the same function code to execute any *function call* such that it accesses its variables by knowing their locations relatively to the **top** of the *stack*, which are the same relative locations to the top of the *activation record* known at *compile time*.

Then, control reaches instruction 1008 again, then calls `factorial(1)` which starts by pushing another *activation record instance* to the *run-time stack*. So, when execution reaches **Position 1** for the **third** time, the *run-time stack* contains:

Return variable	int		
Local variable	int	f	
Local variable	int	r	
Parameter variable	int	n	1
Return address			1012

Return variable	int		
Local variable	int	f	
Local variable	int	r	
Parameter variable	int	n	2
Return address			1012

Return variable	int		
Local variable	int	f	
Local variable	int	r	
Parameter variable	int	n	3
Return address			2008

Then, control reaches instruction 1004 and then **Position 2** for the **first** time, which terminates the call of `factorial(1)`. The compiler saves the return value and return address in registers, then pops the *activation record instance* of `factorial(1)` from the top of the *stack*. The saved return value is assigned to the variable `f` of the *activation record instance* of `factorial(2)`. Then, control resumes from instruction at the saved return address 1012. Now, the *activation record instance* on the top of the *stack* represents the set of variables associated with the `factorial(2)` call, and the *run-time stack* contains:

Return variable	int		
Local variable	int	f	1
Local variable	int	r	
Parameter variable	int	n	2
Return address			1012

Return variable	int		
Local variable	int	f	
Local variable	int	r	
Parameter variable	int	n	3
Return address			2008

Then, control reaches instruction 1016 and the *run-time stack* contains:

Return variable	int		2
Local variable	int	f	1
Local variable	int	r	2
Parameter variable	int	n	2
Return address			1012

Return variable	int		
Local variable	int	f	
Local variable	int	r	
Parameter variable	int	n	3
Return address			2008

Then, control reaches **Position 2** for the **second** time, which terminates the call of `factorial(2)`. The compiler saves the return value and return address in registers, then pops the *activation record instance* of `factorial(2)` from the top of the *stack*. The saved return value is assigned to the variable `f` of the *activation record instance* of `factorial(3)`. Then, control resumes from instruction at the saved return address 1012. Now, the *activation record instance* on the top of the *stack* represents the set of variables associated with the `factorial(3)` call, and the *run-time stack* contains:

Return variable	int		
Local variable	int	f	2
Local variable	int	r	
Parameter variable	int	n	3
Return address			2008

Then, control reaches instruction 1016 and the *run-time stack* contains:

Return variable	int		6
Local variable	int	f	2
Local variable	int	r	6
Parameter variable	int	n	3
Return address			2008

Then, control reaches **Position 2** for the **third** time, which terminates the call of `factorial(3)`. The compiler saves the return value and return address in registers, then pops the *activation record instance* of `factorial(3)` from the top of the *stack*. The saved return value is assigned to the variable `v` of the *activation record instance* of `main()` (we did not show it in the previous figures). Then, control resumes from instruction at the saved return address 2008.

If the *subprogram* contains inner *blocks*, the compiler chooses one of the following two ways to implement its calls:

- Each inner *block* is treated as a call to a *subprogram* with no parameters. In this case, the *activation record* of the *subprogram* does not contain any variable local to an inner *block*. Each inner *block* has its own *activation record*.

- The *activation record* of the *subprograms* contains local variables which are not local to inner *blocks*, and also it contains space sufficient to hold the maximum amount of storage for inner *block* variables at any time during the *subprogram* execution, as shown in the following example:

```
void F(int n)
{
    int x, y, z;
    while(...)
    {
        int a, b, c;
        while(...) { int d, e; }
    }
    while(...) { int f, g; }
}
```

Block variable	int	e
Block variable	int	d
Block variable	int	c
Block variable	int	b and g
Block variable	int	a and f
Local variable	int	z
Local variable	int	y
Local variable	int	x
Parameter variable	int	n
Return address		

5 Simulating recursion

Consider the following *recursive C++* implementation of `MergeSort()`:

```
template<class T>
void Merge(T* a, int istart, int imid, int iend)
{
    T* t=new T[iend-istart];
    int i=istart, j=imid, k=0;
    while(i<imid || j<iend)
    {
        if(i<imid && (j==iend || a[i]<a[j])) t[k++]=a[i++];
        else t[k++]=a[j++];
    }
    for(k=0;k<iend-istart;k++) a[k+istart]=t[k];
    delete[] t;
}
```

```
template<class T> // istart=start_index, iend=last_index+1
void MergeSort(T* a, int istart, int iend)
{
    if(iend-istart<=1) return;
    int imid=(istart+iend)/2;
    MergeSort<T>(a, istart, imid);
    MergeSort<T>(a, imid, iend);
    Merge<T>(a, istart, imid, iend);
}
```



```
template<class T>
void MergeSort(T* a, int n)
{
    MergeSort<T>(a, 0, n);
}
```

```
int main()
{
    int a[]={5,3,7,1,9,2,8,6,4};
    int i, n=sizeof(a)/sizeof(a[0]);
    MergeSort<int>(a, n);
    for(i=0;i<n;i++) cout<<a[i]<<" ";
    return 0;
}
```

Suppose we need to implement `MergeSort()` without *recursion* because of one of the following reasons (recall also the introduction lecture):

- Decrease the usage of the *run-time stack* assigned by the operating system.
- Run the program on embedded system environment which does not support *recursion*.
- Use a programming language which does not support *recursion*.
- Need to avoid using the *run-time stack* to track memory usage in limited-memory environment.
- Make a compiler simulation.

To eliminate *recursion* from any *recursive* function, the following procedure can be applied: (Note that instruction addresses are usually not available to the programmer in high-level programming languages, and we wish to avoid using the `goto` reserved word):

- Push an *encapsulated call* representing the whole *subprogram* to a *stack*.
- Repeat the following: Pop an *encapsulated call* from the *stack*. Execute the *non-recursive* parts. If it contains a *recursive call* (directly or indirectly), do the following:
 - Divide the *called subprogram* into disjoint parts such that each part contains at most one *recursive call*, and each *recursive call* must be the last statement of one part.
 - Encapsulate each *called subprogram* part into an *encapsulated call* object.
 - Push all parts of the *called subprogram* to the *stack* in reverse order.

To implement the `MergeSort()` function without *recursion*, it should be replaced with the following iterative `MergeSortIterative()` function:

```
enum CallType{MERGE_SORT, MERGE};

struct MergeSortCall
{
    CallType type;           // either Merge() or MergeSort()
    int istart, iend;        // used for Merge() and MergeSort()
    int imid;               // used for Merge() only
};
```

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

template<class T>
void MergeSortIterative(T* a, int n)
{
    MergeSortCall stcall;
    stcall.type=MERGE_SORT; stcall.istart=0; stcall.iend=n;

    stack<MergeSortCall> s;
    s.push(stcall);

    while(!s.empty())
    {
        MergeSortCall call=s.top(); s.pop();

        if(call.type==MERGE)
        {
            Merge(a, call.istart, call.imid, call.iend);
            continue;
        }

        // Now, we know that call is for MergeSort()
        if(call.iend-call.istart<=1) continue; // Base case
        int imid=(call.istart+call.iend)/2;

        // MergeSort<T>(a, istart, imid);
        MergeSortCall c1; c1.type=MERGE_SORT;
        c1.istart=call.istart; c1.iend=imid;

        // MergeSort<T>(a, imid, iend);
        MergeSortCall c2; c2.type=MERGE_SORT;
        c2.istart=imid; c2.iend=call.iend;

        // Merge<T>(a, istart, imid, iend);
        MergeSortCall c3; c3.type=MERGE;
        c3.istart=call.istart; c3.imid=imid; c3.iend=call.iend;

        s.push(c3);
        s.push(c2);
        s.push(c1);
    }
}
```