Procedural decomposition. Functions and procedures. Formal and Actual Parameters of a Function. Function return value. Function signature. Passing parameters to a function by value and by reference. Function overloading.

Functions

- **Function** is a named sequence of instructions designed to perform a specific, complete action.
- After a function is executed, control returns to the point where the function was called.
- A function can take parameters and must return a value, which can be empty (`void`). Global variables are also accessible within a function.

Functions

```
type fun_name(type1 arg1,..., typeN argN)
{
      operators;
      [return value;]       // Always required except void
}
```

- `fun_name`: The name of the function. It must be unique within its scope.
- `type`: The return type of the function. It can be void.
- `type1, ..., typeN`: The types of the arguments.
- `arg1, ..., argN`: The names of the arguments.
- `operators`: The operators that make up the body of the function.
- `value`: The value to be returned.

Functions

Function prototype is a declaration that outlines the function without including its body, but specifies the function's return type, name, number and types of arguments. Argument names in a prototype are optional.

```
int sum(int, int);
```

Functions

Function definition (implementation) describes the actions (sequence of instructions) that the function performs. Argument names in the function definition are mandatory.

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

Functions

A function must be declared before it is called. The implementation can be located after the point of the function call.

```
int sum(int a, int b){
    return a + b;
}

int main(){
    int sum_res = sum(1, 2);
    return 0;
}
```

Functions

Often, for improved readability of the program, it is convenient to separate the declaration of function prototypes and their implementations.

```
int sum(int, int);

int main(){
	int sum_res = sum(1, 2);
	return 0;
}
...
int sum(int a, int b){
	return a + b;
}
```

Functions

A function can have its local variables, which are only accessible within the function's body.

```
int math_fun(int a, int b){
    int var1 = (a + b);
    int var2 = (a >> b);

    return var1 + var2;
}
```

Functions

The return of a value from a function is carried out by the return statement. This can involve an implicit type conversion.

```
return expression;
```

The expression whose value will be returned to the point of function call. This expression can be empty if the function's return type is void.

Functions

```
int fun(){
    char a = 0xAA;
    return a;
}

void fun2(int* arr, int num){
    for(int i = 0; i < num; i++)
        if(arr[i] == 5)
            return;
}
```

Functions

The return statement can be omitted in function that does not return a value, indicated by the void return type.

```
void print_args(int a, int b){
    std::cout << “a = ” << a << std::endl;
    std::cout << “b = ” << b << std::endl;
}
```

Passing parameters

When a function is called, the expressions in the place of arguments are evaluated. Memory (typically on the stack) is allocated and initialized for the parameters according to their type. Type conversion is performed if necessary.

Passing parameters

<u>There are several ways to pass parameters to a function in C++:</u>

- By value;
- By pointer (*);
- By reference (&).

Passing parameters

Passing parameters by value

In this method, copies of the argument values are placed on the stack. The function works with these copies. There is no access to the original variables.

Passing parameters

```
int sum(int A, int B){
    A++; B+=3;
    return A + B;
}

int main(){
    int a = 1, b = 2;
    int c = sum(a, b);
}
```

a = 1
b = 2
c = 7

Passing parameters

Passing Parameters by Pointer

Copies of the argument addresses are placed on the stack. Inside the function, there is access to the data located at these addresses. Therefore, the function can modify the values of the arguments.

Passing parameters

```
int sumPtr(int* A, int* B){
    (*A)++; (*B)+=3;
    return *A + *B;
}

int main(){
    int a = 1, b = 2;
    int c = sumPtr(&a, &b);
}
```

a = 2
b = 5
c = 7

Passing parameters

Reference

A reference is an alternative name (alias) for an object.

```
type& refVal = val;
```

A reference can be considered as an additional name for the same memory area.

Passing parameters

- A reference must be explicitly initialized at its declaration, except when it is a function parameter or declared as extern.
- After initialization, a reference cannot be changed to refer to another memory location.
- The type of the reference must match the type of the variable it refers to.
- It's not possible to define references to references, pointers to references, or arrays of references.
- A reference cannot be initialized with a literal.

Passing parameters

```
int a = 1;
int& a_ref = a;

a_ref++;

std::cout << a << std::endl;          2
a++;
std::cout << a_ref << std::endl;      3
```

Passing parameters

Passing Parameters by Reference

Similar to passing by pointer, the addresses of the parameters are passed to the function. Passing a parameter by reference is equivalent to working with the original variable itself, as the argument's name is an alias for the original variable. The function can modify the values of the arguments.

Passing parameters

```
int sumRef(int& A, int& B){
    A++; B+=3;
    return A + B;
}

int main(){
    int a = 1, b = 2;
    int c = sumRef(a, b);
}
```

a = 2
b = 5
c = 7

Passing parameters

Passing parameters by pointer or reference is used for:

- Returning multiple values from a function;
- Passing large data structures to a function to avoid copying data.

Function overloading

Function overloading is a technique where conceptually similar functions, which operate on objects of different types, have the same name. This allows functions to be defined for different types while maintaining a consistent naming convention.

Function overloading

Overloaded functions must differ by the type and/or number of their parameters. Importantly, the return type is not considered in differentiating overloaded functions. This means that functions cannot be overloaded solely based on their return type.

Function overloading

The process of resolving which overloaded function to call follows a specific order:

1. **Exact type match:** The compiler first looks for an overload with an exact match for the types of the arguments.
2. **Type promotion:** If an exact match is not found, the compiler looks for a function where the arguments can be promoted to match the parameters (e.g., short to int, float to double, etc.).
3. **Standard conversions:** If no promotion match is found, standard conversions are considered (e.g., double to int, int to double, pointers to a specific type to void* pointers, and vice versa, etc.).
4. **User-defined conversions:** If none of the above matches are found, the compiler then looks for matches that can be made through user-defined conversions (like custom type casts defined in classes).
5. **Variable number of parameters:** Finally, if no other match is found, the compiler considers functions with a variable number of parameters.

This order of resolution ensures that the most specific version of an overloaded function is chosen to handle the function call, which helps maintain clear and predictable programming behavior.

Function overloading

```
void show(int a){
        std::cout<<“0x”<<std::hex<<a<<std::dec<<std::endl;
}

void show(long long int a){
        std::cout<<“0”<<std::oct<<a<<std::dec<<std::endl;
}

void show(double a){
        std::cout<<std::setprecision(17)<<a<<std::endl;
}

void show(char a){
        std::cout<<“Symbol: ”<<a<<“ Code: ”<<(int)a << std::endl;
}
```

Function overloading

```
int main(){
      show(123);
      show(0xCAFECAFEFF);
      show(0.1 + 0.2);
      show('a');

      short a = 5;
      show(a);
      show((char)57);

      return 0;
}
```

```
0x7b
014537662577377
0.30000000000000004
Symbol: a Code : 97


0x5
Symbol: 9 Code : 57
```

Function overloading

Using a type alias (typedef) does not create a new data type. Therefore, the following function declarations will be equivalent (which will cause a compilation error):

```
typedef unsigned int UINT;
void show(UINT a);
void show(unsigned int a);
```

Function overloading

The NULL macro, representing a null pointer, is defined as 0. Therefore, there can be problems when passing it to functions.

Instead, it is preferable to use the nullptr value.

```
void show(int a){
	std::cout<<“0x”<<std::hex<<a<<std::dec<<std::endl;
}

void show(int* ptr){
	std::cout<<“pointer 0x”<<a<<std::endl;
}
```

Function overloading

```
show(NULL);          0x0
show(nullptr);       pointer: 0x0000000000000000
```

Recursion function

Recursion function

A recursive function is a function that contains a call to itself within its code.

To ensure the computation terminates, it's necessary for the function to have a non-recursive definition for some value of its argument.

Recursion function

```
unsigned long long factorial(unsigned int n){
    if(n > 1)
        return n*factorial(n-1);
    else
        return 1;
}
```

Recursion function

The **depth of recursion** refers to the number of nested calls of the function without returns.

In practice, it's crucial to ensure that the depth of recursion is finite and reasonably small.

Using recursion is generally not recommended because this method of computation can be resource-intensive. Each function call requires additional work with the stack and allocation of memory in it. With a large depth of recursion, this can lead to stack overflow and crash the program.

Functions with a variable number of parameters

Functions with a variable number of parameters are those whose number of parameters is not fixed.

A function can have no mandatory parameters, but such a design might not be convenient to use.

Functions with a variable number of parameters

Explanation:

```
type fun(type1 arg1, type2 arg2, ...);
```

arg1 and arg2 – required parameters.

Functions with a variable number of parameters

Typically, the first parameter contains information about the number of function parameters, or the last parameter contains a stop value, upon encountering which, the function terminates its execution.

Functions with a variable number of parameters

```
unsigned int sum(int n, unsigned int arg1, ...) {
   unsigned int* ptr = &arg1;
   unsigned int res = 0;
   unsigned int inc = 1;

   if((sizeof(void*)/sizeof(unsigned int)) > inc)
      inc = sizeof(void*)/sizeof(unsigned int);

   for(int i = 0; i < n; i++) {
      res += (*ptr);
      ptr += inc;
   }
   return res;
}
```

Functions with a variable number of parameters

```
unsigned int sum2(unsigned int stopVal, ...) {
    unsigned int* ptr = &stopVal;
    unsigned int inc = 1;

    if((sizeof(void*)/sizeof(unsigned int)) > inc)
        inc = sizeof(void*)/sizeof(unsigned int);

    ptr += inc;
    unsigned int res = 0;

    while((*ptr) != stopVal) {
        res += (*ptr);
        ptr += inc;
    }
    return res;
}
```

Functions with a variable number of parameters

```
unsigned res1 = sum(5, 1, 2, 3, 4, 5);       15
unsigned res2 = sum2(5, 1, 2, 3, 4, 5);      10
```

Functions with a variable number of parameters

The same effect of handling a variable number of arguments in a function can be achieved using the macros va_start(), va_arg(), and va_end() from the <cstdarg> library in C++.

- va_start(): This macro is used to initialize the variable argument list. It must be called first before any calls to va_arg(). It takes two arguments: a va_list object to be initialized and the name of the last fixed parameter before the variable argument list begins.
- va_arg(): This macro retrieves the next argument in the parameter list. It takes two arguments: the va_list object and the type of the next argument to be retrieved. It's important to know the expected type of the next argument, as incorrect type specifications can lead to undefined behavior.
- va_end(): This macro ends the processing of the variable argument list. It takes one argument, the va_list object, and performs necessary cleanup. It should always be called before the function returns to avoid resource leaks or other undefined behavior.

Functions with a variable number of parameters

```
unsigned int sum3(int n, unsigned int arg1, ...){
   unsigned int res = 0;
   va_list vl;
   va_start(vl, n);

   for(int i = 0; i < n; i++)
      res += va_arg(vl, int);

   va_end(vl);

   return res;
}
```

Functions with a variable number of parameters

```
unsigned res3 = sum(5, 1, 2, 3, 4, 5);      15
```

Functions with a variable number of parameters

```
int asm_sum(...){
    int* ptr0 = NULL;
    int res = 0;

    __asm {
        mov ptr0, ebp
    }

    int n = *(ptr0 += 2);
    ptr0++;

    for(int i = 0; i < n; i++){
        res += (*ptr0);
        ptr0++;
    }
    return res;
}
```

Functions with a variable number of parameters

```
int res4 = asm_sum(5, 1, 2, 3, 4, 5);        15
```

Function pointers

In C/C++ languages, a function has an address, which is the memory address of its first instruction.

Therefore, this address can be used to initialize a pointer, and the function can be called using this pointer.

Function pointers

Declaration:

```
type (*pointer_name)(type1, type2,...,typeN) = fun;
```

`type` – the return type of the function;

`type1, ..., typeN` – the types of the function's arguments;

`fun` – the name or address of the function to which the pointer is declared.

When working with function pointers, it's convenient to introduce a type alias.

Function pointers

```
bool less(int arg1, int arg2){
    return arg1 <= arg2;
}

bool greater(int arg1, int arg2){
    return arg1 > arg2;
}


bool (*fun_ptr)(int, int) = greater;
bool res = fun_ptr(1, 2);                false
```

Function pointers

When working with function pointers, it is convenient to enter an alias:

```
typedef bool (*CMP_FUN)(int, int);
```

Function pointers

```
void Sort(int* arr, int n, CMP_FUN cmpFun){
      ...
      if(cmpFun(arr[i], arr[j])){
            ...
      }
}

int main(){
      int arr[5] = {5, 2, 3, 4, 1};
      CMP_FUN cmp_fun = greater;
      Sort(arr, 5, cmp_fun);               Sort in ascending order

      cmp_fun = less;
      Sort(arr, 5, cmp_fun);               Sort in descending order
}
```

Function pointers

Array of function pointers:

```
bool greater(int A, int B){...}
bool less(int A, int B) {...}
bool equal(int A, int B) {...}

bool (*fun[])(int, int) = {
                   greater,
                   less,
                   equal
     };
```

Function pointers

```
bool (*fun[3])(int, int) = { 0 };

fun[0] = greater;
fun[1] = less;
fun[2] = equal;
```

Function pointers

```
typedef bool (*FUN_PTR)(int, int);
...

FUN_PTR fun[3] = { 0 };

fun[0] = greater;
fun[1] = less;
fun[2] = equal;
```

Function pointers

Calling functions from an array of function pointers:

```
for (int i = 0; i < 3; i++) {
    std::cout << fun[i](1, 2) << “ ”;
}
```

Callback

Callback (callback function) is the transfer of executable code as one of the parameters of other code.

In C/C++, function pointers are used to work with callback functions.

Callback

```
void Sort(int* arr, int n, CMP_FUN cmpFun)
```

cmpFun – callback function

Callback

Often callback functions are used to handle events:

```
DrObj->MajorFunction[IRP_MJ_READ] = DriverRead;
DrObj->MajorFunction[IRP_MJ_WRITE] = DriverWrite;
DrObj->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DriverDeviceControl;
```

Templates

A **template** is a feature of the C++ language designed for the development of algorithms whose operations are independent of the data types they work on.

The parameter of a template can be any type, including another template. The type must support the operations used by the template function.

Templates

Definition:

```
template<typename T>
type fun(type1 arg1, T arg2, ..., T argN);
```

Instead of the typename keyword, in most cases it is acceptable to use the class keyword.

Templates

There can be several template parameters:

```
template<typename T1, typename T2>
void fun(T1 arg1, T2 arg2)
{
    ...
}
```

Templates

```
template<typename T>
bool greater(T arg1, T arg2){
    return arg1 > arg2;
}

int main(){
    bool res1 = greater<int>(1, 2);
    bool res2 = greater<float>(0.1, 0.2);
}
```

The compiler will create two greater specializations: for int and for float.

Templates

Templates provide a short form for recording a section of source code.

But their use does not shorten the executable code, since for each set of parameters the compiler creates a separate instance of the function.

Templates

Integer constant expressions can be used as template parameters.

```
template<typename T, int N>
void Sort(T* arr, bool (*cmpFun)(T, T)){
      for(int i = 0; i < N; i++){
            ...
      }
}

int main(){
      float arr[5] = {0.1, 0.3, 0.2, 0.5, 0.8};
      Sort<float, 5>(arr, greater);
}
```

Templates

Template functions can have parameters that are only constant expressions:

```
template<int N> void createArray()
{
    int arr[N];
    for(int i = 0; i < N; i++)
        arr[i] = ...
}
```