Functions

Ho Dac Hung

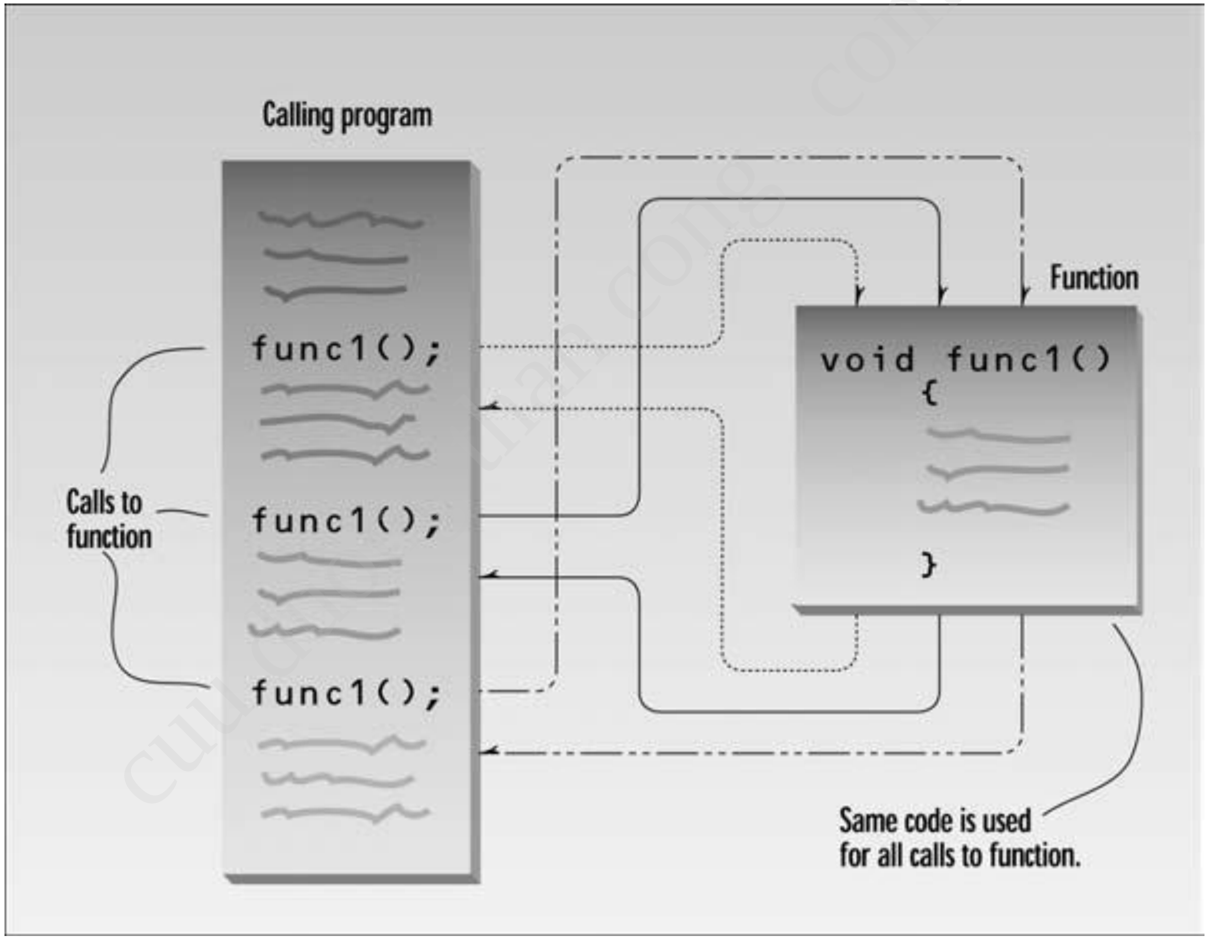
Functions

- A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked from other parts of the program.

Functions

- The most important reason to use functions is to aid in the conceptual organization of a program. Dividing a program into functions is one of the major principles of structured programming.
- Another reason to use functions is to reduce program size. Any sequence of instructions that appears in a program more than once is a candidate for being made into a function.

Functions


Calling program
func1();
func1();
func1();
Calls to function
Function
void func1()
{
}
Same code is used for all calls to function.

```
void starline(); //function declaration
int main()
{
    starline(); //call to function
    return 0;
}
void starline() //function declarator
{
    for(int j=0; j<45; j++) //function body
        cout << '*';
        cout << endl;
}
```

The Function Declaration

- The declaration tells the compiler that at some later point we plan to present a function called starline. The keyword void specifies that the function has no return value, and the empty parentheses indicate that it takes no arguments.

```
void starline();
```

Calling the Function

- The syntax of the call is very similar to that of the declaration, except that the return type is not used. The call is terminated by a semicolon.

```
starline();
```

The Function Definition

- The definition consists of a line called the declarator, followed by the function body. The function body is composed of the statements that make up the function, delimited by braces.
- The declarator must agree with the declaration: It must use the same function name, have the same argument types in the same order (if there are arguments), and have the same return type.

Comparison with Library Functions

- The declaration is in the header file specified at the beginning of the. The is in a library file that’s linked automatically to your program when you build it.
- When we use a library function we don’t need to write the declaration or definition. But when we write our own functions, the declaration and definition are part of our source file.

Eliminating the Declaration

- The second approach to inserting a function into a program is to eliminate the function declaration and place the function definition (the function itself) in the listing before the first call to the function.

```
void starline()
{
    for(int j=0; j<45; j++)
        cout << '*';
        cout << endl;
}

int main() //main() follows function
{
    starline();
    return 0;
}
```

Passing Arguments to Functions

- An argument is a piece of data passed from a program to the function. Arguments allow a function to operate with different values, or even to do different things, depending on the requirements of the program calling it.

Passing Arguments to Functions

- Constants
- Variables
- Structures

Passing Arguments to Functions

```
void repchar(char ch, int n) //function declarator
{
    for(int j=0; j<n; j++) //function body
        cout << ch;
        cout << endl;
}
```

Passing Arguments to Functions

```
int main()
{
    char chin;
    int nin;
    cout << “Enter a character: “;
    cin >> chin;
    cout << “Enter number of times to repeat it: “;
    cin >> nin;
    repchar(chin, nin);
    return 0;
}
```

Passing Arguments to Functions

- Passing by value
- Passing by reference

Returning Values from Functions

- When a function completes its execution, it can return a single value to the calling program. Usually this return value consists of an answer to the problem the function has solved.

Returning Values from Functions

```
float lbstokg(float pounds)
{
    float kilograms = 0.453592 * pounds;
    return kilograms;
}
```

Returning Values from Functions

```
int main()
{
    float lbs, kgs;
    cout << “\nEnter your weight in pounds: “;
    cin >> lbs;
    kgs = lbstokg(lbs);
    cout << “Your weight in kilograms is “ << kgs <<
        endl;
    return 0;
}
```

Overloaded Functions

- An overloaded function appears to perform different activities depending on the kind of data sent to it. Overloading is like the joke about the famous scientist who insisted that the thermos bottle was the greatest invention of all time.

Overloaded Functions

- Different Numbers of Arguments

```
void repchar(); //declarations
void repchar(char);
void repchar(char, int);
```

Overloaded Functions

- Different Kinds of Arguments

```
void engldisp( Distance ); //declarations
void engldisp( float );
```

Recursion

- The existence of functions makes possible a programming technique called recursion. Recursion involves a function calling itself.

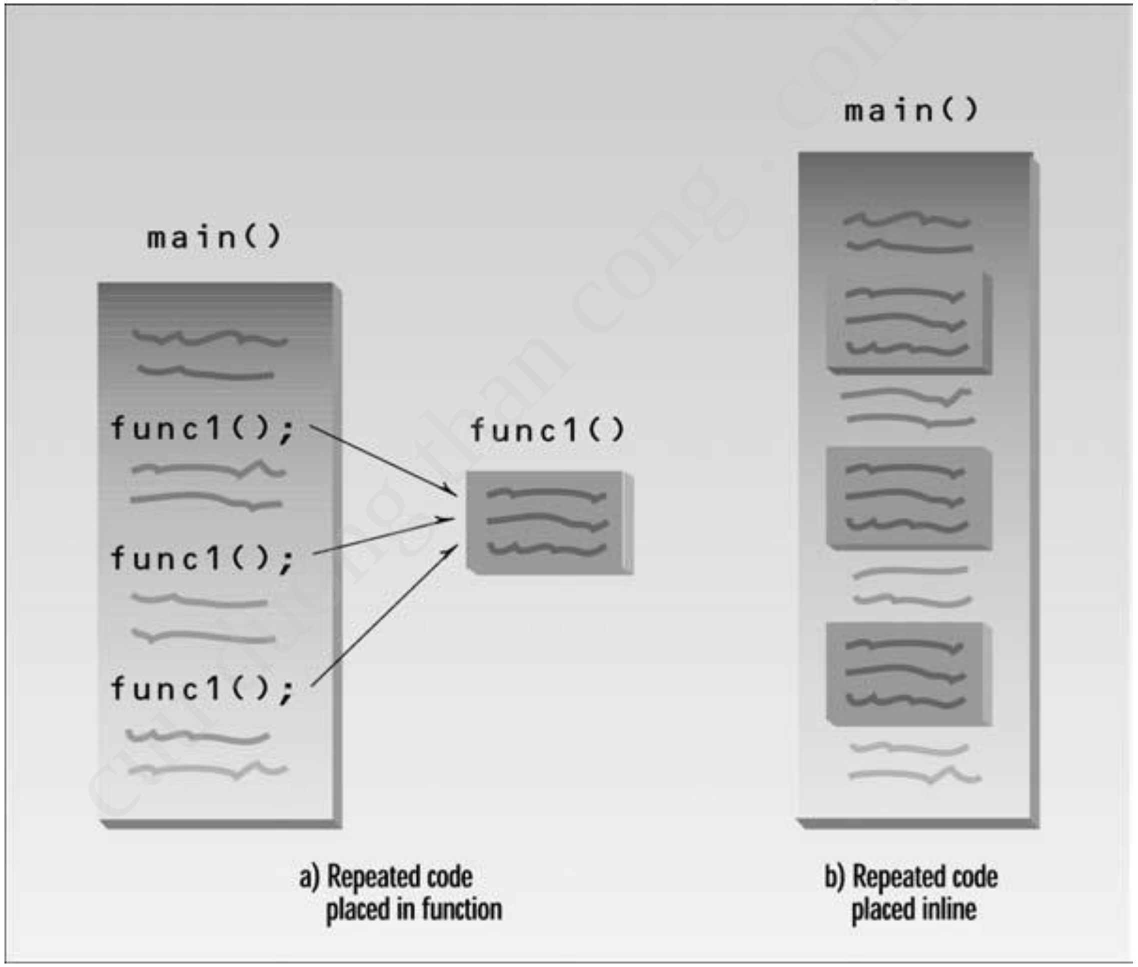
Recursion

```
unsigned long factfunc(unsigned long n)
{
    if(n > 1)
        return n * factfunc(n-1); //self call
    else
        return 1;
}
```

Inline Functions

- We mentioned that functions save memory space because all the calls to the function cause the same code to be executed; the function body need not be duplicated in memory.
- To save execution time in short functions, you may elect to put the code in the function body directly inline with the code in the calling program. That is, each time there’s a function call in the source file, the actual code from the function is inserted, instead of a jump to the function.

Inline Functions


main()
func1();
func1();
func1();
func1()
main()
a) Repeated code placed in function
b) Repeated code placed inline

Inline Functions

```
inline float lbstokg(float pounds)
{
    return 0.453592 * pounds;
}
```

Default Arguments

- Surprisingly, a function can be called without specifying all its arguments. This won’t work on just any function: The function declaration must provide default values for those arguments that are not specified.

Default Arguments

```
void repchar(char=’*’, int=45);

int main()
{
    repchar(); //prints 45 asterisks
    repchar(‘=’); //prints 45 equal signs
    repchar(‘+’, 30); //prints 30 plus signs
    return 0;
}
```

Local Variables

- Variables defined within a function body are called local variables because they have local scope.

Global Variables

- Global variables are defined outside of any function. A global variable is visible to all the functions in a file.
- Global variables have storage class static, which means they exist for the life of the program. Memory space is set aside for them when the program begins, and continues to exist until the program ends.

Static Local Variables

- A static local variable has the visibility of an automatic local variable (that is, inside the function containing it). However, its lifetime is the same as that of a global variable, except that it doesn’t come into existence until the first call to the function containing it. Thereafter it remains in existence for the life of the program.

Static Local Variables

```
float getavg(float newdata)
{
    static float total = 0;
    static int count = 0;
    count++;
    total += newdata;
    return total / count;
}
```

Returning by Reference

- Besides passing values by reference, you can also return a value by reference. One reason is to avoid copying a large object. Another reason is to allow you to use a function call on the left side of the equal sign.

```
int x;
int& setx();

int main()
{
    setx() = 92;
    cout << “x=” << x << endl;
    return 0;
}

int& setx()
{
    return x;
}
```

const Function Arguments

- Suppose you want to pass an argument by reference for efficiency, but not only do you want the function not to modify it, you want a guarantee that the function cannot modify it.

const Function Arguments

```
void aFunc(int& a, const int& b);

int main()
{
    int alpha = 7;
    int beta = 11;
    aFunc(alpha, beta);
    return 0;
}
```