Fundamentals of Programming 1

Functions

Arkadiusz Chrobot

Department of Information Systems

November 25, 2022

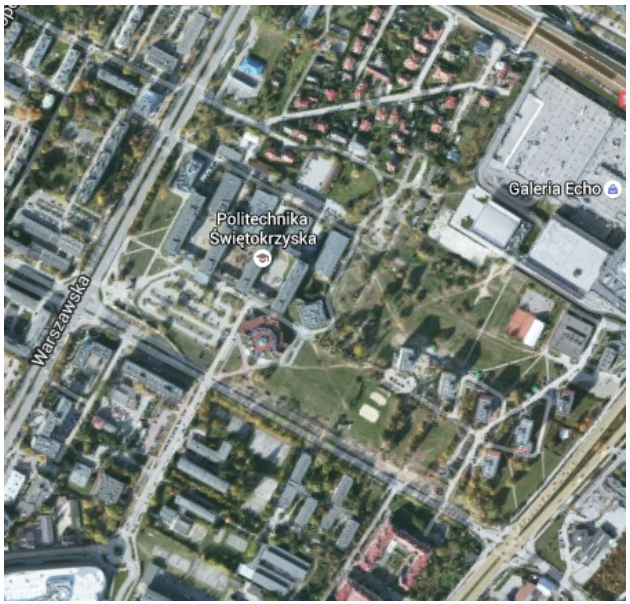
Outline

Procedural Programming

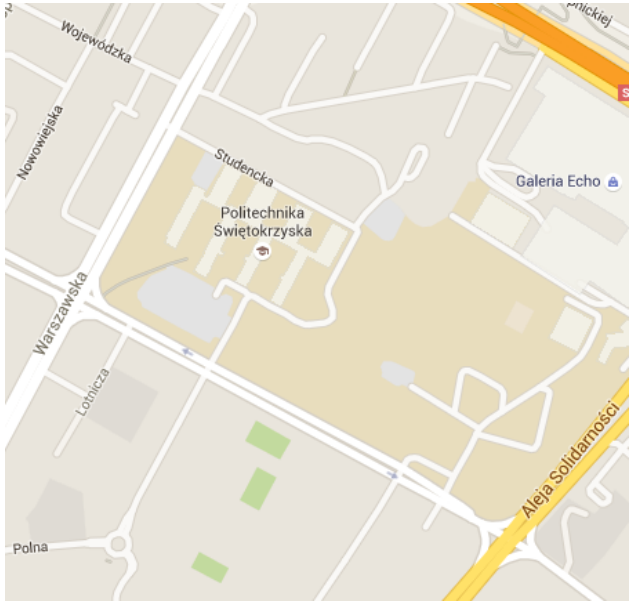
One of the principles of the procedural programming paradigm is a simplification of a program. The simplification is obtained by partitioning the source code into small units, each with a unique name. Those units are organized into a hierarchy, which means that a unit may use other units provided they are defined or declared ahead. This allows programmers to apply the Cartesian analysis method known from mathematics to programming problems. Such a problem can be solved by dividing it into small subproblems that are easy to solve. The solution of the original problem is achieved by combining solutions of all those subproblems. The procedural programming paradigm also allows for using an abstraction.

Abstraction

The abstraction in programming (and in other disciplines) is a process of simplifying a problem by highlighting those parts of its definition that are vital for its solution and hiding those which are not important. Maps are an example of application of the abstraction:



(a) A satellite image



(b) A map

Source: Google Maps

Functions

In the C language functions are an implementation of the concept of *subroutines*. They allow the programmer to group and give a name to the statements that together perform a specific task. Functions provide a way for creating new statements, that are more suitable for solving a given problem. Those statements are build on statements that already exist in a programming language. In other words, functions make it possible to use the abstraction. They also let programmers reuse the code. Finally, a function may return a value, which makes it similar to the mathematical concept of the same name.

Functions

Function Definition

The overall structure of a function *definition* is as follows:

```
returned_value_data_type function's_name(parameters_list)
{
    ...
}
```

The first line is called a *prototype* or a function's *header*. It begins with the declaration of the data type of the returned value, followed by a function's identifier (a name). After the name a list of parameters is declared in the parentheses. The parameters are special variables that allow the function to exchange data with other parts of the program. The parameters list is followed by a block that is called a function's *body* or a *text*.

Functions

Returned Value

A function may return a value. For this purpose the `return` keyword has to be used inside the function's body. The keyword must be followed by an expression whose value is returned. The expression may be optionally enclosed in parentheses. It can be complex or very simple, like a single variable, a constant or even a literal. The value of the expression has to be compatible with the data type of the returned value, declared in the function's header. The `return` keyword is also an exit point for the function.

Functions

Function's Call

To perform its task, the function has to be *called* (*invoked*) from within a definition of another function, like for example the `main()` function[1]. The *invocation* of a function is made by placing its name followed by the parentheses in the code. If the function doesn't have any parameters then the parentheses should be empty. Otherwise they ought to contain *arguments* that are substituted for the parameters. There should be as many arguments as parameters and the types of arguments should be compatible with the types of corresponding parameters. After the function exits (finishes its job) the flow of control returns to the statement that follows the function's call in the source code. If the function returns a value, then the value may be assigned to a variable of a compatible type. Such a function may also be called inside an expression.

[1] Please notice, that I put the parentheses after the function's name to distinguish it in the text from the name of a variable or any other element of the code.

Functions

The `void` Keyword Usage

A function is an implementation of an algorithm. Some algorithms do not require input data, so functions that implement them need neither arguments nor parameters. To indicate that the function has no parameters, inside its definition, the `void` keyword can be used in the list of parameters. Some programmers tend to leave the list's parentheses empty. This is not the same as using the `void` keyword. It means that the function can take an unspecified number of arguments. The `void` keyword may also be used to indicate, that the function does not return any value. In that case it should be applied as the return value data type. The same keyword can be used to indicate that the program should ignore the value returned by the function. To this end it is placed in parentheses before a function call. It literally means that the type of the value is cast to void. However, such a notation is rarely used. It is sufficient to invoke the function without assigning its returned value to any variable. This means that the program performs the function for its *side effects*.

Functions

First Simple Example

```c
#include<stdio.h>

int f1(void)
{
        puts("I'm the f1() function and I return 5.");
        return(5);
}

int variable_1, variable_2;

int main(void)
{
        variable_1 = f1();
        variable_2 = 5*f1();
        (void) f1();
        f1();
        return 0;
}
```

Functions

Comment to the First Example

The `f1()` function in the program from the previous slide takes no arguments but prints a message on the screen and returns 5. It is invoked 4 times in the program. The first time its value is assigned to the `variable_1`. The second time it is invoked in an expression. The value the function returns is multiplied by 5 and the result is assigned to the `variable_2`. In the last two calls the value returned by the function is ignored, but the message is displayed on the screen. Please notice, that the name of the function is very simple. This is sufficient for the example, but in more advanced program a descriptive name should be given to the function. Such an identifier should contain at least one verb, to indicate what the function does. Please also notice, that the value placed after the `return` keyword could be in parentheses (the `f1()` function) or not (the `main()` function).

Functions

Second Simple Example

```c
#include<stdio.h>

int f1()
{
        puts("I'm the f1() function and I return 5.");
        return(5);
}

int variable_1, variable_2;

int main(void)
{
        variable_1 = f1(1,2,3,4,5,6,7);
        variable_2 = 5*f1(variable_1, variable_2);
        return 0;
}
```

Functions

Comment to the Second Example

The example shows the difference between a definition of a function in which the list of parameters is empty and the one in which the list of parameters contains the `void` keyword. In the former case the list of parameters is seemingly empty, but it is possible to pass to the function any number of arguments, that will never be used. That can lead to possible errors.

Functions

Third Simple Example

```c
#include<stdio.h>

void f2(void)
{
        puts("I'm f2() function and I return nothing.");
}

int a;

int main(void)
{
        f2();
        /*a = f2();*/ // This is not allowed.
        return 0;
}
```

Functions

Comment to the Third Example

In the third example a function is defined that returns no value. It only prints a message on the screen. Such a function cannot be used in an expression. Its value cannot be assigned to a variable, because it simply does not exist.

Functions

Forth Simple Example

```c
#include<stdio.h>

void f3(void)
{
        puts("I'm the f3() function.");
        return;
        puts("I return nothing and I won't print this message.");
}

int main(void)
{
        f3();
        return 0;
}
```

Functions

Comment to the Forth Example

Strange as it may seem, it is possible to use the `return` keyword in a function that returns nothing. However, it must be immediately followed by a semicolon. In this case the `return` keyword simply terminates the function. Any statements that follow it won't be performed.

Functions

A Declaration of a Function

A declaration of a function consists only of its header and a semicolon that follows. A function declared in such a way has to be defined in other part of the program, but the declaration allows the programmer to use the function before it is defined. The declaration of a function has many applications. One of them is reversing the hierarchy of definitions of functions. That makes it possible to read them in the more natural "top-bottom" way. Such an effect can be achieved by declaring first all the necessary functions and then defining the `main()` function and the earlier declared functions. The function declaration can be applied when the programmer wants to use a function, but for some reason she/he cannot declare it before using.

Functions

A Declaration of a Function — An Example

```c
#include<stdio.h>

void f4(void); //A function's prototype.

int main(void)
{
        f4();
        return 0;
}

void f4(void)
{
        puts("I'm the f4() function and I was defined after the main() function.");
}
```

Local variables

Until now all variables that we used in programs have had a global scope. The C programming language permits to declare variables inside a function or even at the beginning of a block. Starting from the ISO C99 standard, it lets the programmer to declare a variable just before it should be used. It makes the program more legible. Applying in the program local variables has many pros. Local variables allow for better memory usage than the global ones. The former exist in memory only when the function in which they are declared is performed. For that reasons they are also known in the C language as the *automatic* variables. Moreover, local variables are only visible in the block in which they are declared, starting from the place of their declaration. They are invisible outside the block, but the statements inside the block can access all variables (and other elements of code) declared outside.

Local variables

Variable Shadowing

Local variables can be given the same names as the global variables or even other local variables declared in different blocks. When there are many variables declared in the program which have the same name, in a specific block under that name is available only the one with the nearest declaration. The others are outside the scope of that block. This is called a *variable shadowing*. That also applies to the names of all elements of a program and in that case it is called a *name masking*. For example a local constant of the same name as global variable is masking that variable.

Local Variables

Memory Allocation And Deallocation For Local Variables

The allocation and deallocation of the memory for a local variable happens automatically and, like in the case of global variables, is transparent to the programmer. The only thing she or he has to do is to declare such a variable. However, the local variables exist only when the function in which they are declared is executed. This is possible because the memory area of a program is divided into several parts. One of them (a code segment) stores all the instructions that are to be executed during a program run. The next one (a data segment) holds a place for all global variables. Another one is a call stack segment. Whenever a function is invoked, a *stack frame* also known as an *activation record* is created on the call stack for this instance (invocation) of the function. In the frame there is a place for local variables, parameters (which are a form of local variables) and an address of return (the address of a statement that should be performed after the function exits).

Local Variables

Memory Allocation And Deallocation For Local Variables

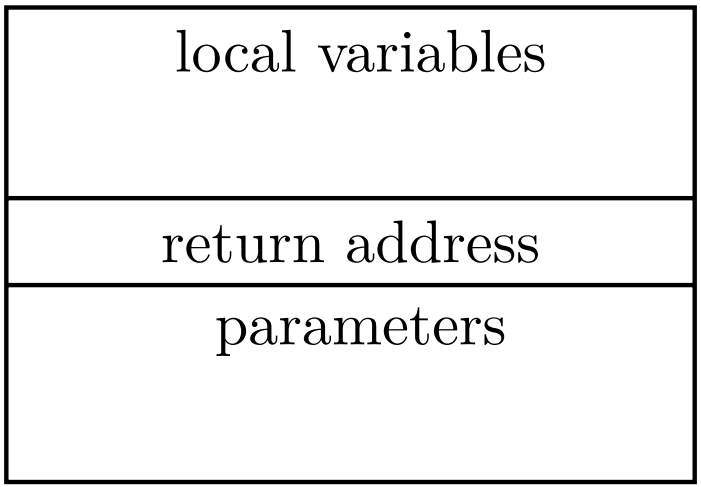


Figure: Stack frame — a sketch

Local Variables

Memory Allocation And Deallocation For Local Variables

The call stack behaves according to the Last In First Out (LIFO) rule. When a function calls another function then a stack frame for the called function is created on the top of the calling function's activation record. That frame is also destroyed before the calling function's frame is removed, what follows the order in which those functions complete. If the calling function invokes another function then the area of the call stack that was occupied by the stack frame of the previously called function may be reused for the activation record of the newly called function. **As a consequence all local variables have an unspecified initial value and they must be initialized by the programmer.** If a variable is declared inside a block that is a part of a function, the memory for that variable is also allocated in the function's stack frame.

Local Variables

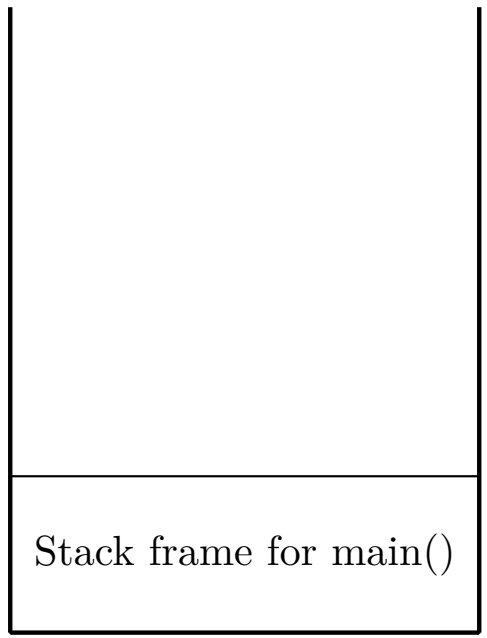
Memory Allocation And Deallocation For Local Variables

```c
#include<stdio.h>

void f2(void)
{
}

void f1(void)
{
        f2();
}

int main(void)
{
        f1();
        return 0;
}
```



The stack after program launch

Local Variables

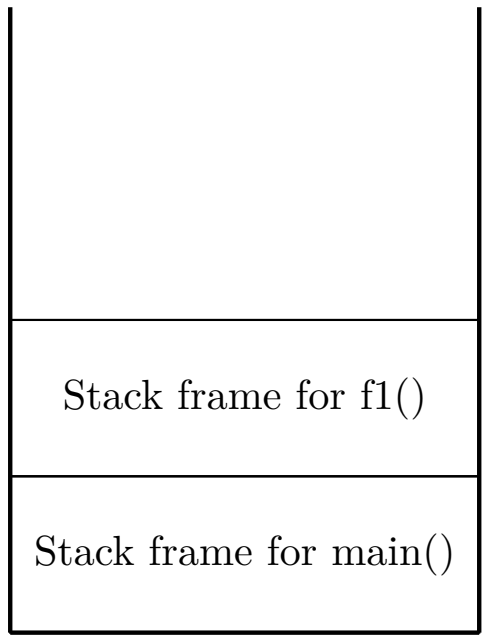
Memory Allocation And Deallocation For Local Variables

```
#include<stdio.h>

void f2(void)
{
}

void f1(void)
{
        f2();
}

int main(void)
{
        f1();
        return 0;
}
```



The stack after invoking the `f1()` function

Local Variables

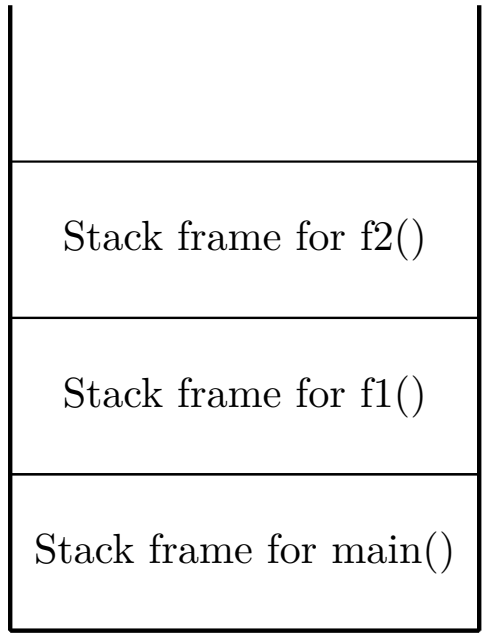
Memory Allocation And Deallocation For Local Variables

```
#include<stdio.h>

void f2(void)
{
}

void f1(void)
{
        f2();
}

int main(void)
{
        f1();
        return 0;
}
```



The stack after invoking the `f2()` function

Local Variables

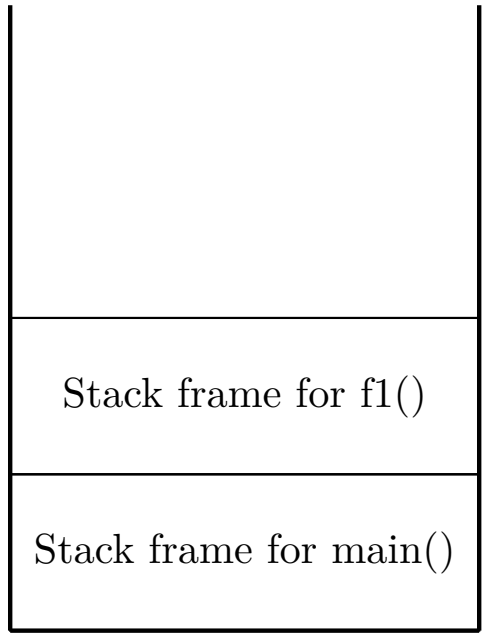
Memory Allocation And Deallocation For Local Variables

```
#include<stdio.h>

void f2(void)
{
}

void f1(void)
{
        f2();
}

int main(void)
{
        f1();
        return 0;
}
```



The stack after the `f2()` has completed

Local Variables

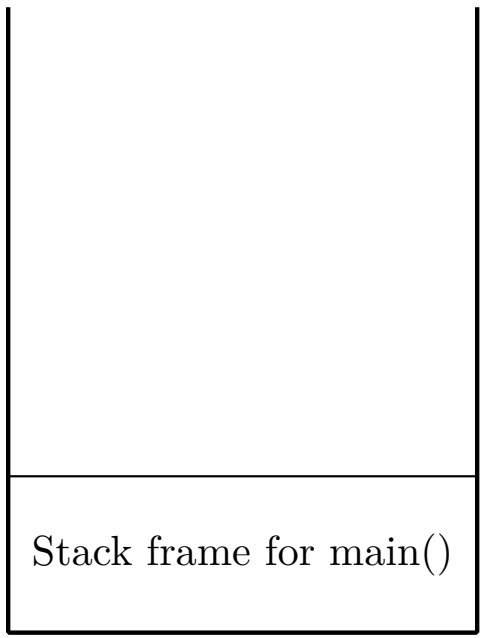
Memory Allocation And Deallocation For Local Variables

```
#include<stdio.h>

void f2(void)
{
}

void f1(void)
{
        f2();
}

int main(void)
{
        f1();
        return 0;
}
```



The stack after the `f1()` has completed

Local Variables

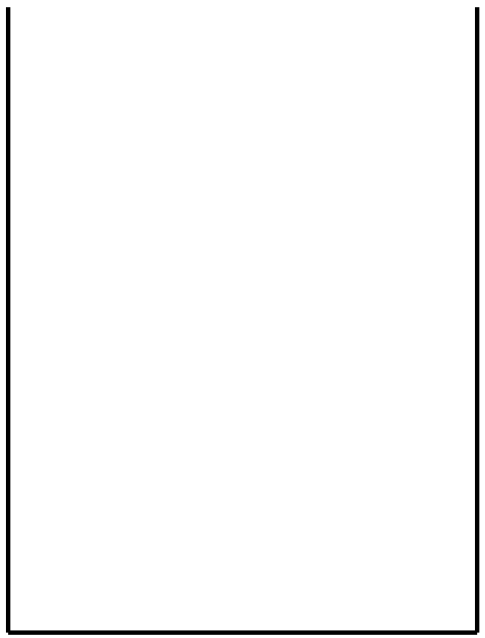
Memory Allocation And Deallocation For Local Variables

```c
#include<stdio.h>

void f2(void)
{
}

void f1(void)
{
        f2();
}

int main(void)
{
        f1();
        return 0;
}
```



The stack after the program ends

Local Variables

Memory Allocation And Deallocation For Local Variables

```c
#include<stdio.h>

void f2(void)
{
        int x = 1000000;
        printf("x = %d\n", x);
}

void f1(void)
{
        short int a;
        short int b;
        printf("a = %d\n", a);
        printf("b = %d\n", b);
}
```

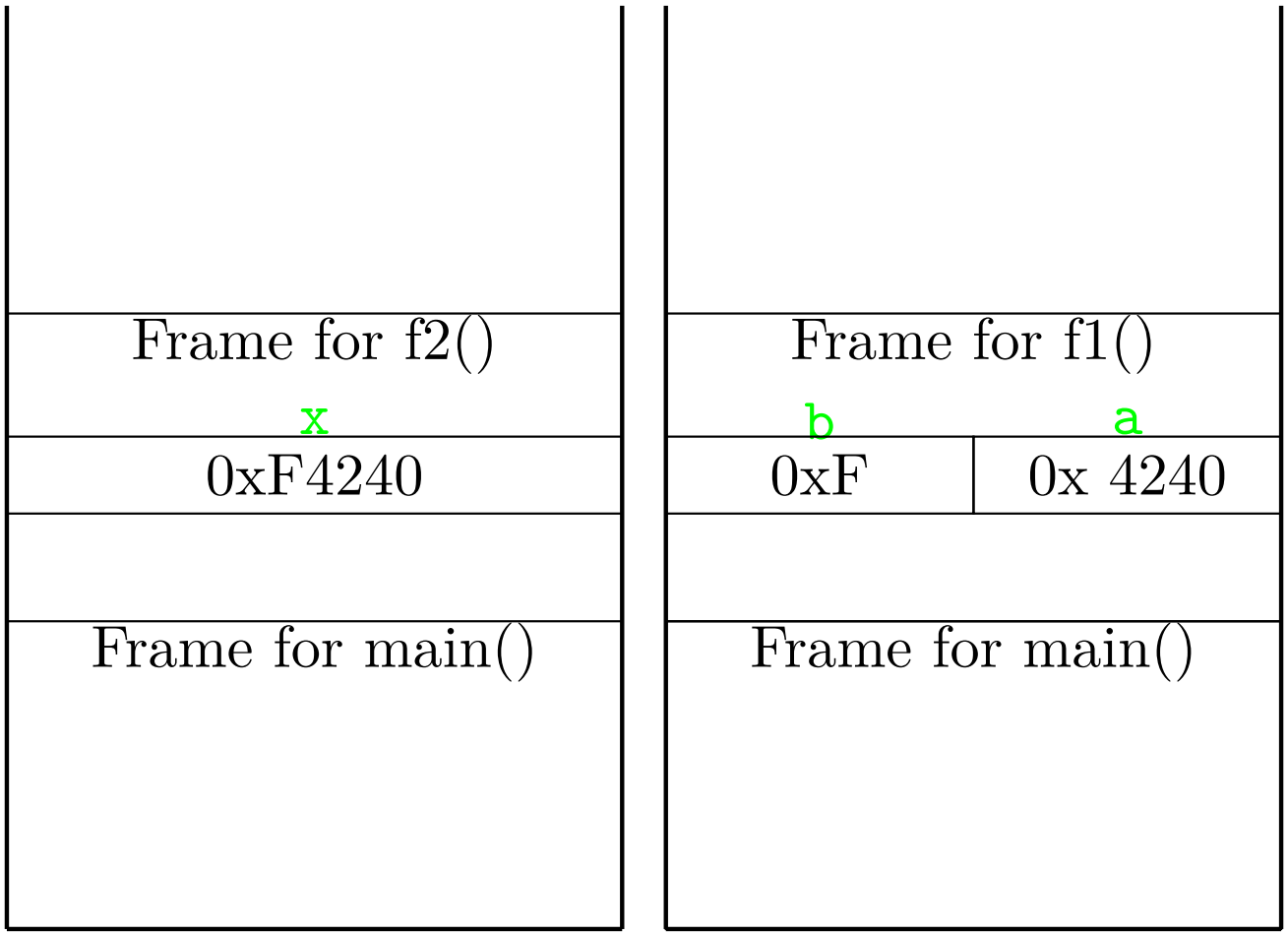
Local Variables

Memory Allocation And Deallocation For Local Variables

```
int main(void)
{
        f2();
        f1();
        return 0;
}
```

Local Variables

Memory Allocation And Deallocation For Local Variables



(a) For The f2() Function (b) For The f1() Function

Figure: How The Frames Are Created on The Stack

Local Variables

The `static` Keyword

Taking into the consideration the properties of local variables it is recommended to use them instead of the global variables whenever it is feasible. The `static` keyword when applied to a local variable extends its lifetime to the entire run-time of a program in which the variable is declared. Moreover, that variable has an initial value of zero and is still accessible only from within the function's body. When one of the function's instances modifies the value of the variable, then the subsequent invocations will notice the modification. In other words, a local variable declared with the `static` keyword has the characteristics of both the local and the global variable.

Local Variables

An Example

```c
#include<stdio.h>

void count_instances(void)
{
    static unsigned int sum;
    sum+=1;
    printf("The function was invoked %d times.\n",sum);
    int i;
    for(i=0; i<5; i++) {
        int i = 1;
        printf("This \"i\" variable isn't a loop counter: %d\n",i);
    }
}

int main(void)
{
    count_instances();
    count_instances();
    return 0;
}
```

Direct Usage of Global Variables

Functions may directly reference global variables but it is not the best practise. It has many disadvantages. Let's try to answer the question: what does the function invoked in the following code snippet do?

Invocation of `add_numbers()` Function

```
int sum = add_numbers();
```

Judging by its name, it adds some numbers. The type of the variable to which is assigned the result of that function suggests that those numbers are integers. However, we are unable to deduce how many those numbers are, because they are stored in global variables. If we wanted to use the function in another program we would have to copy not only the definition of the function but also the declarations of global variables used by the function. That's only two of the disadvantages.

Parameters

To avoid issues described in the previous slide the parameters may be applied. A parameter is a special local variable that allows the function to exchange data with the rest of the program. A function may have more than one parameter. The parameters must have different names than the local variables declared directly in the function's body (outside of other blocks in the body). Each parameter may have a unique type or the types may repeat. In the location of code where the function is invoked the parameters must be substituted by arguments of compatible types. There should be as many arguments as parameters. There are three ways of passing an argument by a parameter. Parameters make functions more universal.

Passing By Value

Parameters that pass arguments by value are declared similarly to regular variables, but in the parameters list of a function. Their declarations are separated by commas. When more than one parameter of the same type should be declared then the type of such a parameter must be repeated in each declaration. All the declarations can be placed in one line. The arguments passed by such parameters can be literals, constants, local and global variables and even expressions. Those parameters are input parameters. If a variable is passed by such a parameter, then any modification made to the value of the parameter inside a function body does not affect the value of the variable. It is possible to use parameters that pass arguments by value together with other kinds of parameters.

Passing By Value

An Example

```c
#include <stdio.h>

void f5(int x, int y)
{
    puts("Inside the function:");
    printf("The value of \"x\" parameter before a change: %d\n", x);
    x+=1;
    printf("The value of \"x\" parameter after the change: %d\n", x);
    printf("The value of \"y\" parameter: %d\n",y);
}

int main(void)
{
    int a=3;
    printf("The value of \"a\" variable before passing to f5() function: %d\n",a);
    f5(a,2*a);
    printf("The value of \"a\" variable after f5() function finishes: %d\n",a);
    return 0;
}
```

Passing By Value

Comment to the Example

After running the program and reading all messages displayed on the screen it can be noticed that modification of the `x` parameter had no effect on the value of the `a` variable. Parameters and arguments that are substituted by them may have the same names. Passing by value in the example is equivalent to the following assignments:

```
int x = a;
int y = 2*a;
```

Passing By Constant

If for some reason the value of the parameter should not be modified in the function then passing by constant may be applied. The declaration of such a parameter is prefixed with the `const` keyword. The same kinds of arguments may be passed by such a parameter as in the case of passing by value. Parameters passing by constant may be used together with parameters passing by value.

Passing By Constant

An Example

```c
#include <stdio.h>

void f6(const int x)
{
    printf("The value of \"x\" parameter: %d\n",x);
    printf("The value of an expression with the \"x\" parameter: %d\n",x+1);
    /* x+=1; */ // It is not allowed. It won't even compile.
}

int a = 3;

int main(void)
{
    printf("The value of \"a\" variable before passing to the function: %d\n",a);
    f6(a);
    printf("The value of \"a\" variable after the function finishes: %d\n",a);
    return 0;
}
```

Passing By Constant

Comment to the Example

As it might be expected, the value of the parameter is not modified in the example. Passing by constant is equivalent to the following assignment:

```
const int x = a;
```

Introduction to Pointers

Before a third way of passing argument by parameters will be introduced we are going to learn about a new type of variable. Such a variable is called a `pointer`. It is declared according to the following pattern:

```
type_of_variable *name_of_variable;
```

The `*` (a star or an asterisk) is the only thing that differentiates pointer declaration from the declaration of a regular variable. If we used the `sizeof` operator to measure the size of such a variable we would learn that it is always 4 bytes for 32-bit computers and 8 bytes for 64-bit computer. In other words, its size is independent of the data type used in its declaration. This is because, the pointer does not store a value directly, but it stores an address of a variable, called the pointed variable, that stores the value. The data type in the pointer declaration determines what type of variables can be pointed by the pointer.

Introduction to Pointers

A neutral value of the pointer is defined by the `NULL` constant, however newer editions of the C language standard allow programmers to use `0` in the place of this constant. To assign an address of a variable to the pointer the address operator has to be applied. It is represented by the `&` (an ampersand) symbol. To read a value of a variable pointed by the pointer a dereference operator has to be applied. It is represented by the `*` (a star or an asterisk) symbol. If the address stored in the pointer should be displayed on the screen the `"%p"` conversion specifier for the `printf()` function should be used. The address is displayed as a hexadecimal number.

Introduction to Pointers

An Example

```c
#include <stdio.h>

int main(void)
{
  int *pointer = NULL;
  int variable = 3;
  pointer = &variable;
  printf("The value of the pointed variable: %d\n",*pointer);
  printf("The address stored in the pointer: %p\n",pointer);
  variable++;
  printf("The value of the pointed variable: %d\n",*pointer);
  *pointer+=1;
  printf("The value of the pointed variable: %d\n",variable);
  return 0;
}
```

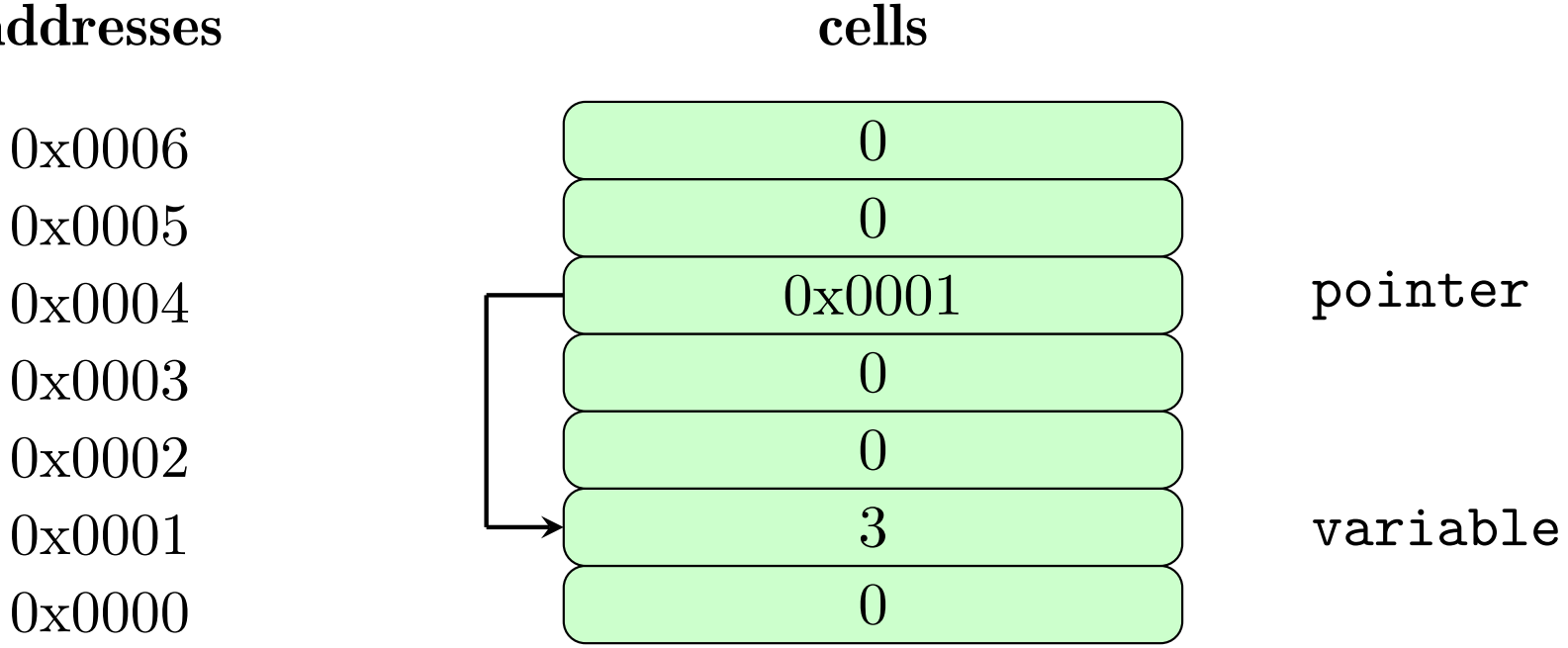
Introduction to Pointers

Comment to the Example

If we run and traced the program with the use of a debugger, we would see that the value of the pointed variable can be both modified and read with the use of the pointer. Please notice the difference between reading the value of the pointer (the address that it stores) and the value of the pointed variable.

Introduction to Pointers

For better understanding of the pointers let's take a look at the very simplified model of the Random Access Memory (RAM) in which every variable has the size of a single memory cell. The variables from the example program could be placed in such a memory as follows:



Passing By Pointer

The pointers may be used as parameters of a function. Arguments for such parameters may be only compatible pointers or addresses of compatible variables acquired with the use of the address operator. The pointer parameter is both input and *output* parameter. The function may pass by the parameter results to the rest of the program. It means that the function may return more than one value with the use of such parameters. It is possible to use passing by pointer parameters together with passing by value and passing by constant parameters.

Passing By Pointer

An Example

```c
#include <stdio.h>

void f7(int *x)
{
    puts("Inside the function:");
    printf("The value of \"*x\" before a change: %d\n",*x);
    *x+=1;
    printf("The value of \"*x\" after the change: %d\n",*x);
}

int main(void)
{
    int a = 3;
    printf("The value of \"a\" variable before the f7() function call: %d\n",a);
    f7(&a);
    printf("The value of \"a\" variable after f7() function finishes: %d\n",a);
    return 0;
}
```

Passing By Pointer

Comment to the Example

Please observe how the `f7()` functions is invoked and how the dereference operator `*` is used. In this example passing by pointer could be replaced by passing by value and returning a result by the function. However, such a replacement is not possible if a function returns more than one value.

Pure Functions

The concept of a *pure function* is one of the principles of functional programming. A pure function is a function that does not have any side effects and only returns a single value (a result). It means that the function does not change directly or indirectly the value of any global variable (the global state). Such functions are very useful in concurrent programming, when threads are applied, because that functions do not affect the state of threads others than those which invoked them and thus are safe to use in such cases.

Exit From a Function

A place in code where the function terminates is called an *exit point*. The procedural paradigm requires that only one exit point exists in the function. It means that the `return` keyword should be used only once in the function body and in case of functions that do not return a value it shouldn't be used at all. However, it is common among C programmers to disobey the rule, because using the `return` keyword multiple times in function body makes it short, simpler, easier to understand and often more efficient.

Recommendations

1. A function declaration should be short and legible.
2. A function should have a descriptive name containing a verb.
3. A function should have at least one parameter. On the other hand it shouldn't have too many parameters.
4. A function should do only one task described by its name.
5. Functions without parameters should be used scarcely.
6. A function should **never** use global variables directly.

The Unix programmers follow a convention of creating functions, that is often applied by other programmers. By the convention a function returns as a value an integer that indicates the state of completing the function's task. If it is zero, then the task was completed successfully. If it is negative it indicates failure and the absolute value of the integer usually identifies the exception that caused the problem. The function passes its results with the help of pointer parameters.

Quadratic Equation — a Version With Functions

A Function That Takes Equation's Coefficients From The User

```c
#include<stdio.h>
#include<math.h>

void get_abc_coefficients(float *a, float *b, float *c)
{
    puts("Please enter the coefficients of the quadratic equation:");
    do {
        printf("a= ");
        scanf("%f",a);
        if(*a==0.0)
            puts("The vale of \"a\" coefficient mustn't be zero!");
    } while(*a==0.0);
    printf("b= ");
    scanf("%f",b);
    printf("c= ");
    scanf("%f",c);
}
```

Quadratic Equation — a Version With Functions

Comment to the Function

The function is responsible for assigning to its parameters the values of coefficients entered by the user. It does only this task. Please notice, that inside the function the second argument passed to the `scanf()` invocation is not prefixed with an ampersand. This is because, the `scanf()` function takes as the second argument an address of a variable and the parameters `a, b` and `c` are pointers that store such addresses. Using the address operator in their cases would be an error, because the operator would return addresses of the pointers instead of addresses of variables pointed by them.
The included header files are not a part of the function, but their presence is required to compile the whole program successfully.

Quadratic Equation — a Version With Functions

A Discriminant Calculating Function

```
float calculate_delta(float a, float b, float c)
{
    return b*b-4*a*c;
}
```

Quadratic Equation — a Version With Functions

A Function Implementing the *signum* Operation

```
int signum(float number)
{
    return (number<0) ? -1 : 1;
}
```

Quadratic Equation — a Version With Functions

A Function Calculating the q Coefficient

```
float calculate_q(float b, float delta)
{
    return -0.5*(b+signum(b)*sqrt(delta));
}
```

Quadratic Equation — a Version With Functions

A Function Calculating a Single Root of the Quadratic Equation

```
float calculate_root(float a, float q)
{
    return q/a;
}
```

Quadratic Equation — a Version With Functions

A Function Calculating Two Roots of the Quadratic Equation

```
void calculate_roots(float a, float c, float q, float *x1, float *x2)
{
    *x1=q/a;
    *x2=c/q;
}
```

Quadratic Equation — a Version With Functions

The `main()` function

```
int main(void)
{
    float a=0.0,b=0.0,c=0.0;
    get_abc_coefficients(&a,&b,&c);
    float delta = calculate_delta(a,b,c);
    if(delta==0.0) {
        float q = calculate_q(b,delta);
        printf("The equation has a single root %.10f\n",
                calculate_root(a,q));
     }
     if(delta>0.0) {
            float q = calculate_q(b,delta);
            float x1=0.0, x2=0.0;
            calculate_roots(a,c,q,&x1,&x2);
            printf("The functions has two roots - x1:\
            %.10f, x2: %.10f\n",x1,x2);
     }
     if(delta<0.0)
        puts("The equation doesn't have real roots.");

    return 0;
}
```

Quadratic Equation — a Version With Functions

Comment to the Example

Please notice, that one of the strings in the `main()` function is divided into two with the use of the backslash (\) character. It's a special character that informs the compiler that those two strings are in reality one. The program with the functions is longer than the original program demonstrated in the previous lecture. However, it is easier to understand. It is also easier to distinguish between the cases when the quadratic equation has one root or two roots if the program is partitioned into functions.

Thanks

Many thanks to Grzegorz Łukawski, PhD and Leszek Ciopiński, MSc for helping me to complete the Polish version of these slides.

Questions

?

The End

Thank You for Your attention!