Imperative programming

Dynamic program structure

Tamás Kozsik et al

Eötvös Loránd University

October 19, 2022

Table of contents

Dynamic program structure

- How does the program work?
- Information about the status of program execution
- Subroutine calls from the main program
- Storing variables in memory
- Abstract model

Functions

- Sub-programs
  - Functions
  - Procedures
  - Routines
  - Methods
- Resolution of larger programs
- Parameterization
- Calling (evaluation)
- Returning result

Functions

Simple function

$$even : \mathbb{Z} \to \mathbb{L}$$

$$even(x) = \begin{cases} \uparrow & \text{if } x \text{ is even} \\ \downarrow & \text{if } x \text{ is odd} \end{cases}$$

```
bool even(int x)
{
  return x % 2 == 0;
}

int main()
{
  if (even(42))
    printf("Even\n");
  else
    printf("Odd\n");
}
```

Functions

Without return value

```
bool goodArgs(int argc, char* argv[]) { ... }

void printUsage(char programName[]) {
  printf("This program can be used as follows:\n");
  printf("%s --in <input> --out <output>", programName);
}

int main(int argc, char* argv[]) {
  if (!goodArgs(argc, argv)) {
    printUsage(argv[0]);
    return 1;
  }

  doTheJob();

  return 0;
}
```

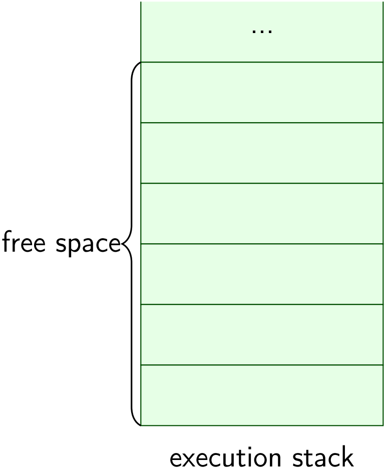
Execution stack

```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```

- Execution stack
- Logic of sub-program calls
  - LIFO: Last-In-First-Out
  - Stack data structure
- An entry about every sub-program call
  - Activation record
  - E.g. information about where to return
- Bottom of stack: main program's activation record
- Top of stack: where program execution is

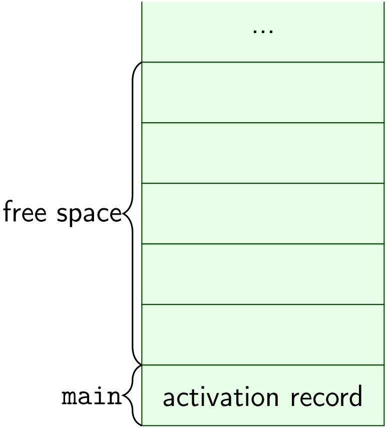
Managing sub-program calls

```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



Managing sub-program calls

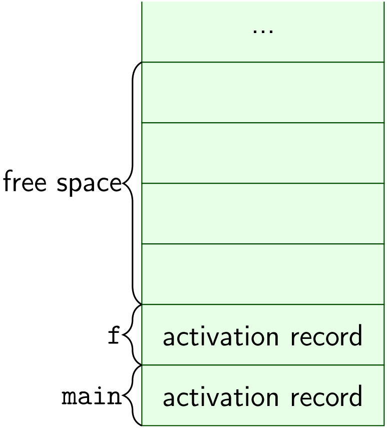
```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



execution stack

Managing sub-program calls

```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



...

free space

f

activation record

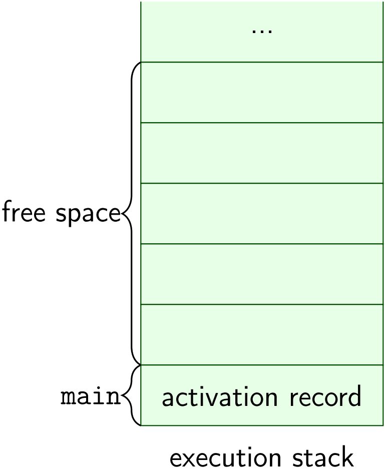
main

activation record

execution stack

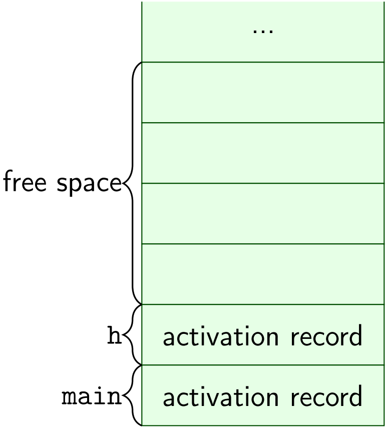
Managing sub-program calls

```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



Managing sub-program calls

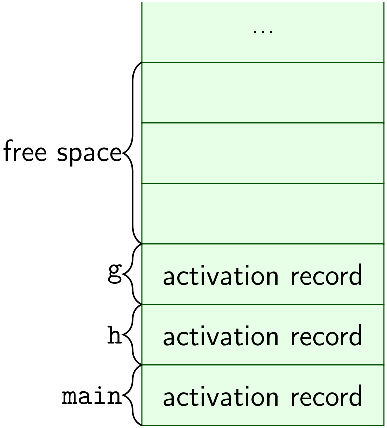
```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



execution stack

Managing sub-program calls

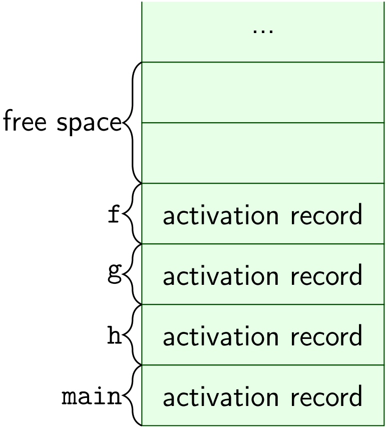
```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



execution stack

Managing sub-program calls

```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



...

free space

f activation record

g activation record

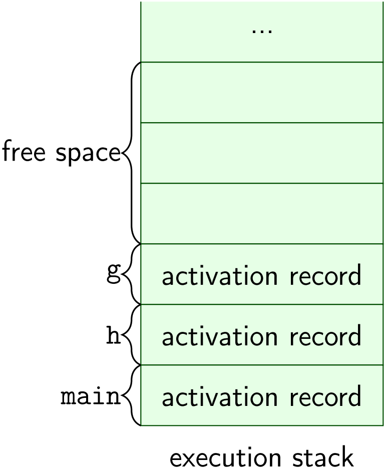
h activation record

main activation record

execution stack

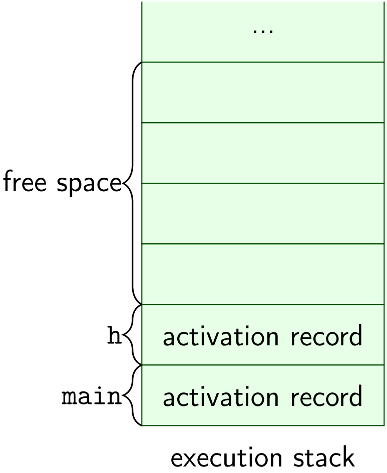
Alprogramhívások nyilvántartása

```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



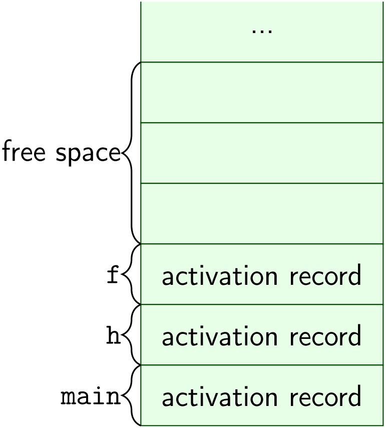
Managing sub-program calls

```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



Managing sub-program calls

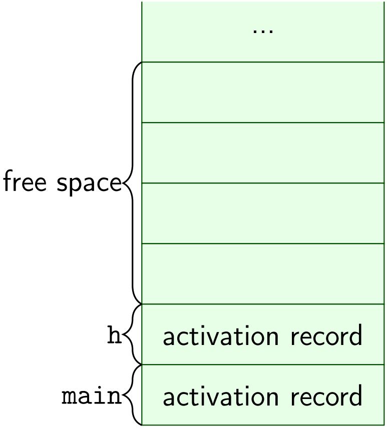
```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



execution stack

Managing sub-program calls

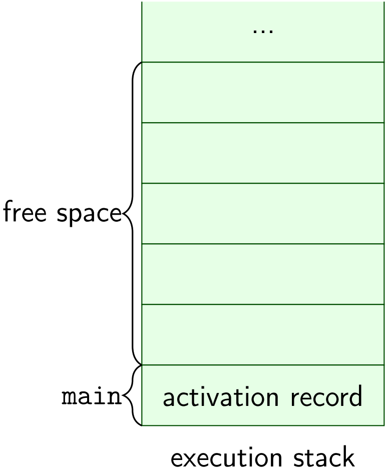
```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



execution stack

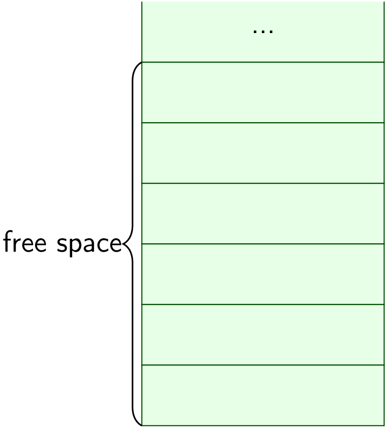
Managing sub-program calls

```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```



Managing sub-program calls

```
void f()
{
}
void g()
{
  f();
}
void h()
{
  g();
  f();
}
int main()
{
  f();
  h();
}
```

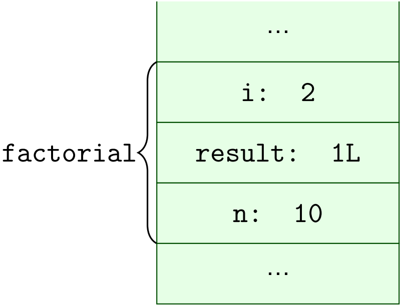


execution stack

Execution stack

- All kinds of technical things
- Sub-program parameters
- (Some) local variables of sub-program

```
long factorial(int n)
{
  long result = 1L;
  int i = 2;
  for (; i <= n; ++i)
    result *= i;
  return result;
}
```



execution stack

Recursion

- A function calls itself
  - Directly
  - Indirectly
- New activation record for each call
- Too deep recursion: Stack Overflow
- Cost: building and destructing activation records

Factorial

```
int factorial(int n) {
  if (n < 2)
    return 1;
  else
    return n * factorial(n - 1);
}
```

```
int factorial(int n) {
  return n < 2 ? 1 : n * factorial(n - 1);
}
```

```
int factorial(int n) {
  int result = 1;
  for (int i = 2; i <= n; ++i)
    result *= i;
  return result;
}
```

Tail-recursion

Obvious

```
int factorial(int n) {
  return n < 2 ? 1 : n * factorial(n - 1);
}
```

Tail-recursive

```
int fact_acc(int n, int acc) {
  return n < 2 ? acc : fact_acc(n - 1, n * acc);
}

int fact(int n) {
  return fact_acc(n, 1);
}
```

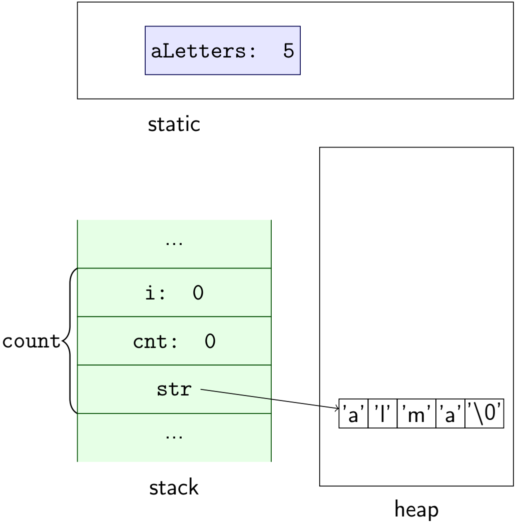
Storing variables in memory

- Execution stack → automatic
- Static storage → static
- Dynamic storage → dynamic

Variable definition

- With declaration
  - Static and automatic storage
    - Static storage
    - Execution stack
  - Lifetime: from program structure
    - The scope
    - Except for local static
- With allocation statement
  - Dynamic storage
    - Heap (Dynamic store)
  - Lifetime: programmable
  - Deallocation
    - Deallocation statement (C, C++)
    - Garbage collection (Haskell, Python, Java)

stack - static - heap



aLetters: 5

static

...

i: 0

cnt: 0

str

...

count

stack

'a' 'l' 'm' 'a' '\0'

heap

```
int aLetters = 0;
int count(char* str)
{
  int cnt = 0, i = 0;
  while (str[i] != '\0')
  {
    if (str[i] == 'a')
      ++cnt;
    ++i;
  }
  a_letters += cnt;
  return cnt;
}
```

Automaic storage variable

- In the execution stack (in activation records)
- Local variables are *usually* like this
- Lifetime: block execution

```
int lnko(int a, int b) {
  while (b != 0) {
    int c = a % b;
    a = b;
    b = c;
  }
  return a;
}
```

Static storage variable

- Static storage
  - Static declaration evaluation
  - Compiler knows how big storage is needed
- E.g. global variables
- Lifetime: from the beginning of the execution to the end

```
int counter = 0;
int signal()
{
  return ++counter;
}
```

Static local variables

- static keyword
- Scope: local variable (data hiding)
- Lifetime: like global variables

```
int counter = 0;
int signal()
{
  return ++counter;
}
```

```
int signal()
{
  static int counter = 0;
  return ++counter;
}
```

Sub-program parameters

- In the definition: formal parameter list
- At the call: actual parameter list

```
void f(int x) {
  return 2 * x;
}

int main() {
  printf("42 twice: %d\n", f(42));
}
```

Parameter passing techniques

- pass-by-value, call-by-value
- call-by-value-result
- call-by-result
- call-by-reference
- call-by-sharing
- call-by-need
- call-by-name

Parameter passing by value

- Formal parameter: automatic storage local variable
- Actual parameter: initial value
- Call: Actual parameter's value is copied to the formal parameter
- Return: the formal parameter is deallocated

Parameter passing by value

Example

```
int gcd(int a, int b)
{
  while (b != 0) {
    int c = a % b;
    a = b;
    b = c;
  }
  return a;
}

int main()
{
  int n = 1984, m = 356;
  int r = gcd(n, m);
  printf("%d %d %d\n", n, m, r);
}
```