Lecture 14

Code generation part 1

Runtime environment

Hyosu Kim

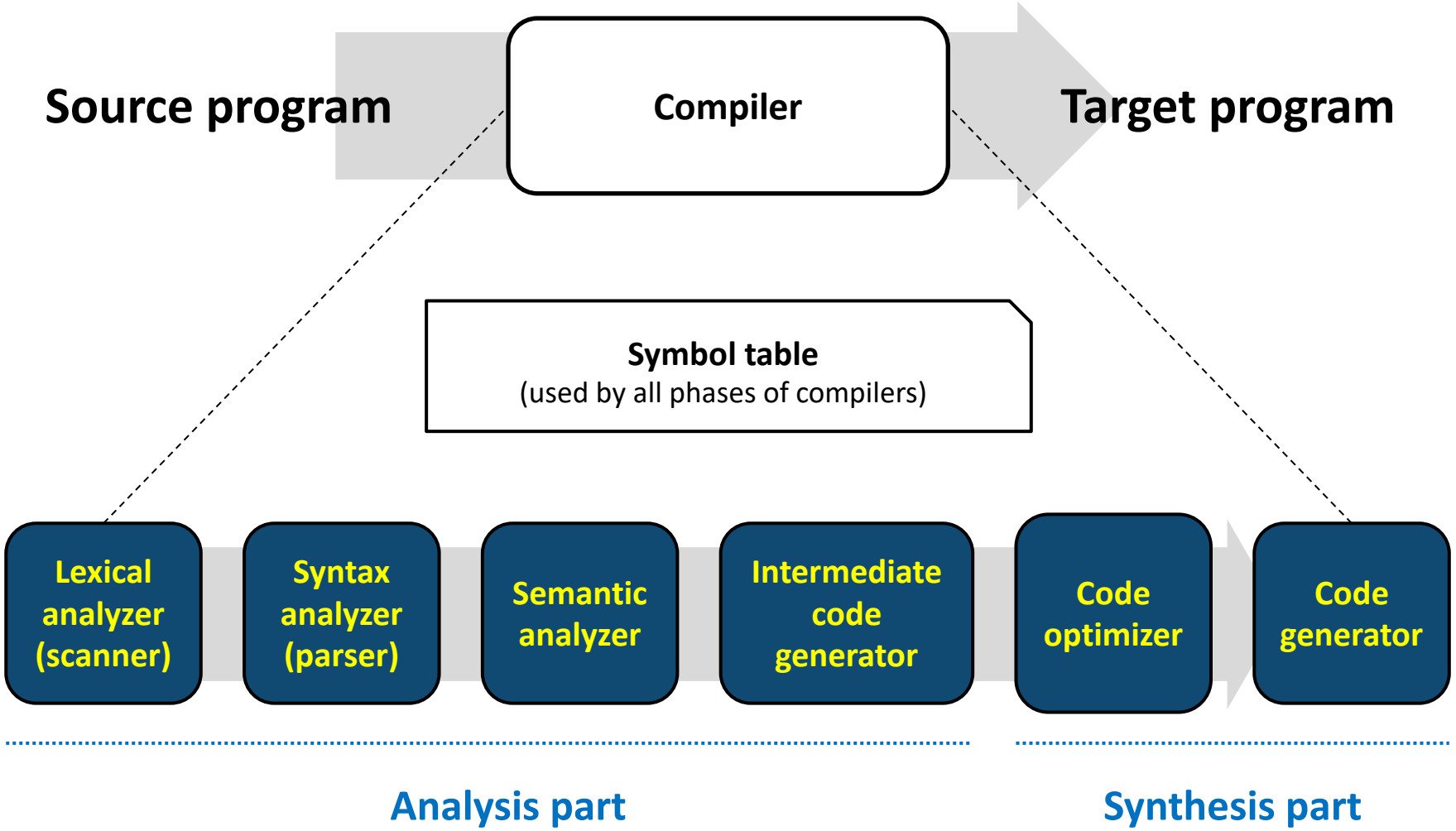
School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://sites.google.com/view/hyosukim

hskimhello@cau.ac.kr, hskim.hello@gmail.com

Overview


Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part

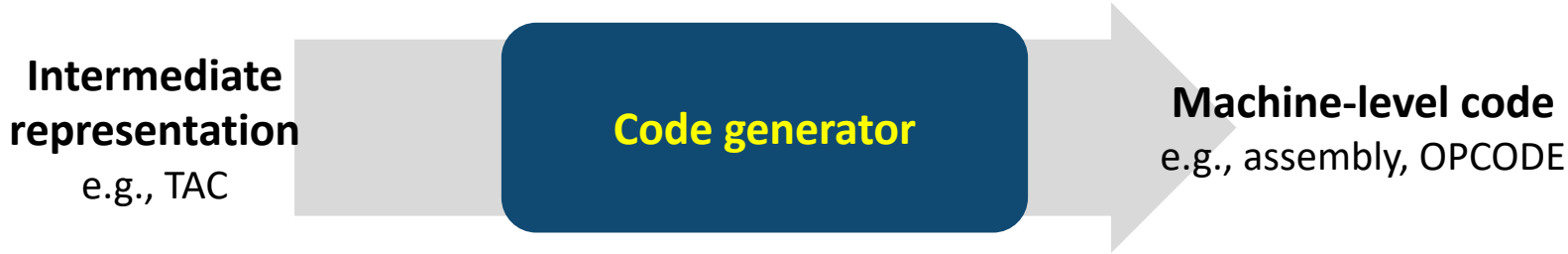
Code generator

Translates an intermediate representation (e.g., three address code)

into a machine-level code (e.g., assembly code)

Code generator

Before discussing code generation



- Data copy operations
- Arithmetic operations
- Comparison operators
- Control jumps
- **Objects**

  (primitive type variables, arrays, …)
- **Function calls**
- **Parameter passing**

Q. How to support such high-level structures in machine-level code???

Runtime environment!!

- Data copy operations
- Arithmetic operations
- Comparison operators
- Control jumps

Runtime environment

A set of data structures used at runtime to support high-level structures

This environment deals with a variety of issues

- **What do objects look like in memory?**

  The representation of the objects in memory space

- **What do functions look like in memory?**

  The linkages between functions

  The mechanisms passing parameters

- **Where in memory should objects and functions be placed?**

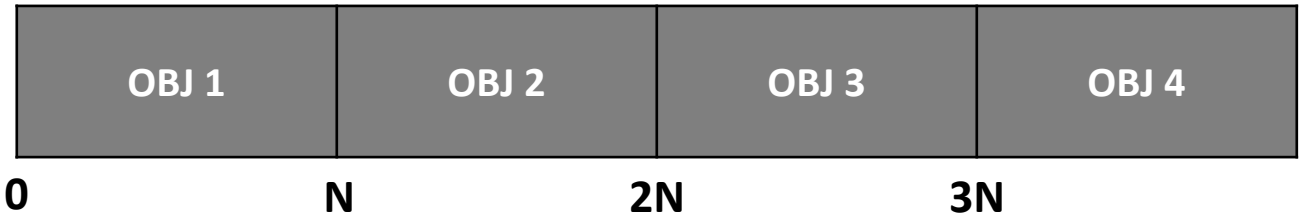
  The layout and allocation of memory space for the objects and functions

Object representations

Data alignment

Compilers determine how objects are arranged & accessed in computer memory

- Objects are **N-byte aligned** in memory space
  - N-byte alignment: the start memory address of objects is a multiple of N bytes



  - N can be different depending on the type of objects

Object representations

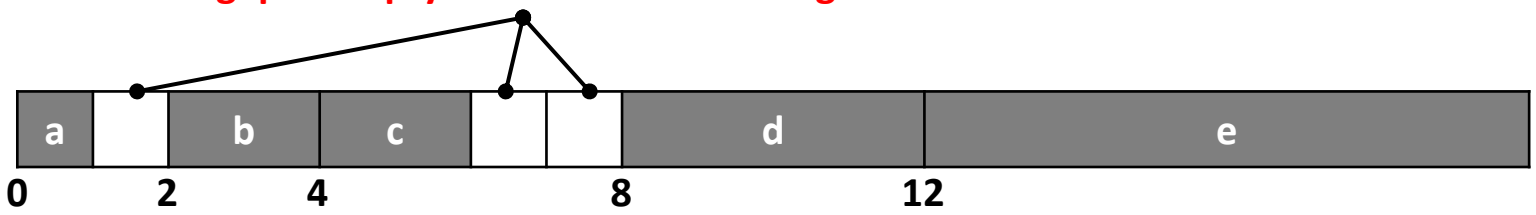
Data alignment

Example: typical data structure alignment of C objects in x86 (32-bit) machines

Primitive type	Size	Alignment
char	1-byte	1-byte
short	2-byte	2-byte
int	4-byte	4-byte
long	4-byte	4-byte
float	4-byte	4-byte
double	8-byte	4-byte
long long	8-byte	4-byte
Any type of pointers	4-byte	4-byte

```
char a;
short b;
short c;
int d;
double e;
```

Padding: put empty characters for the alignment



Object representations

Representing arrays

In different programming languages, arrays are differently represented in memory space

For an array A[n],

- C-style arrays: each element A[i] is stored consecutively in memory space

  (aligned based on its base type)

A[0]	A[1]	A[2]	A[3]	…	A[n-1]

- Java-style arrays: each element A[i] is stored consecutively in memory space

  + size information is prepended

n	A[0]	A[1]	A[2]	A[3]	…	A[n-1]

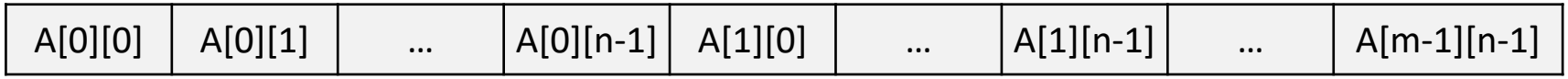
Object representations

Representing multi-dimensional arrays

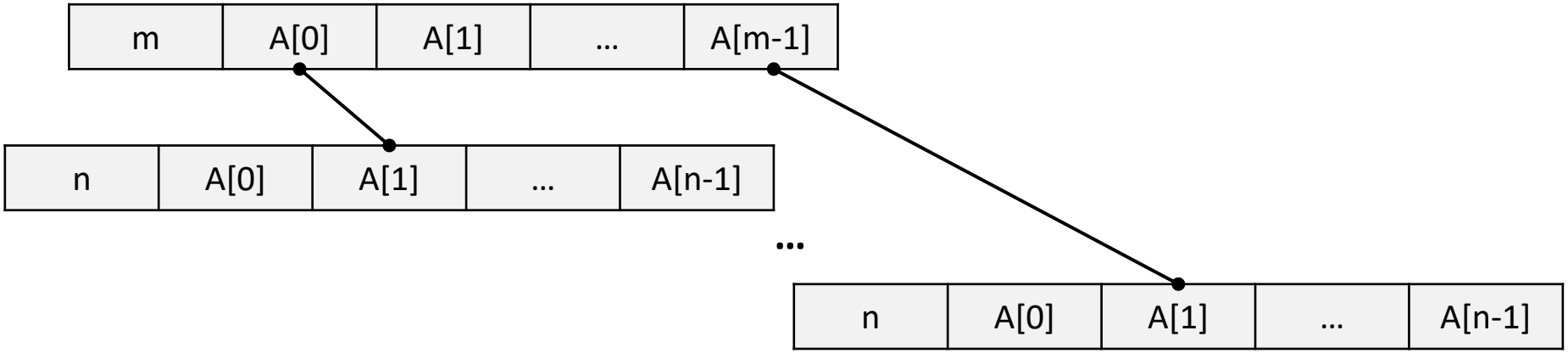
Often represented as an array of arrays

For an array A[m][n],

- C-style arrays



- Java-style arrays



Runtime environment

A set of data structures used at runtime to support high-level structures

This environment deals with a variety of issues

- **What do objects look like in memory?**

  The representation of the objects in memory space

- **What do functions look like in memory?**

  The linkages between functions

  The mechanisms passing parameters

- **Where in memory should objects and functions be placed?**

  The layout and allocation of memory space for the objects and functions

Function representations

Two main questions related with function representations

1) How to represent the relationship between functions?

(e.g., function calls)

2) How to keep the information needed to manage functions?

(e.g., parameters, return values, return address, …)

Activations

An invocation of a function F is called an activation of F

- The lifetime of an activation of F is until all the statements in F is executed

- F can invoke another function Q

  Then, the lifetime of an activation of F includes the execution of all the statements in Q

- The relationship between function activations can be depicted as a tree,

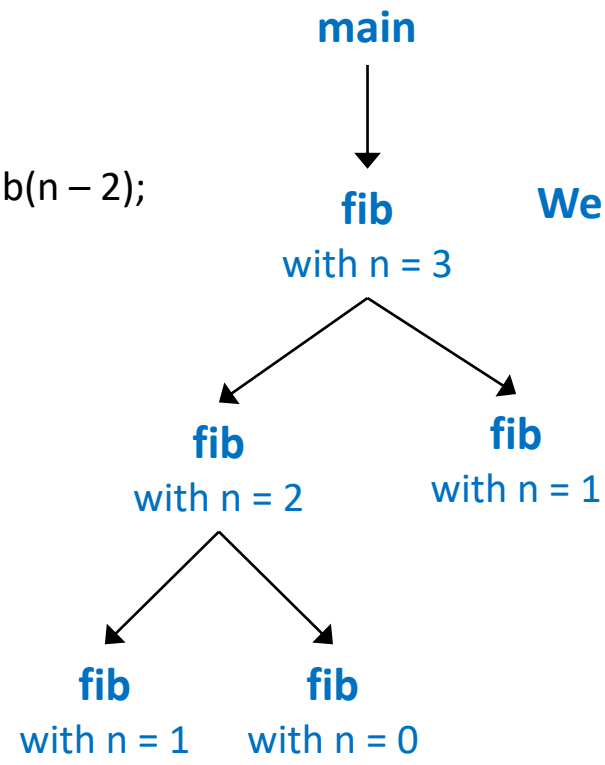
  called an **activation tree**

Activations

An example of activation trees

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   fib(3);
}
```



Activations are properly nested!!

We can manage an activation of functions by using a stack!!

Activations

An example of activation trees

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   fib(3);
}
```

main

main() is called

[stack]

Currently active function: main

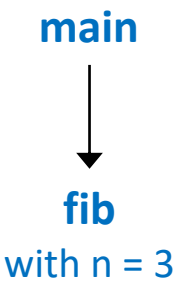
Pending functions: none

Activations

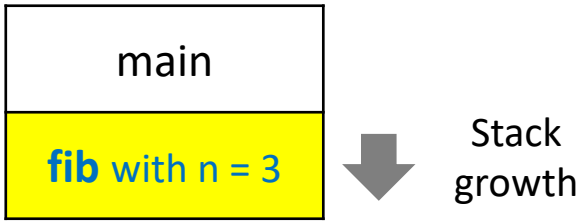
An example of activation trees

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   fib(3);
}
```



fib(3) is called in main()



[stack]

Currently active function: fib with n = 3

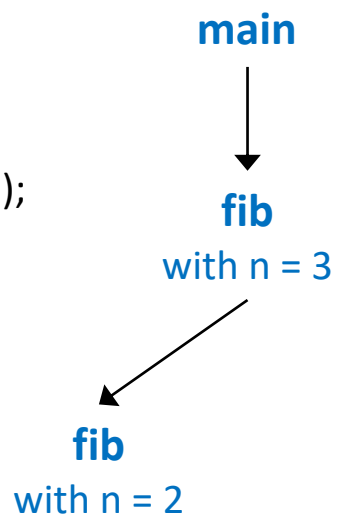
Pending functions: main

Activations

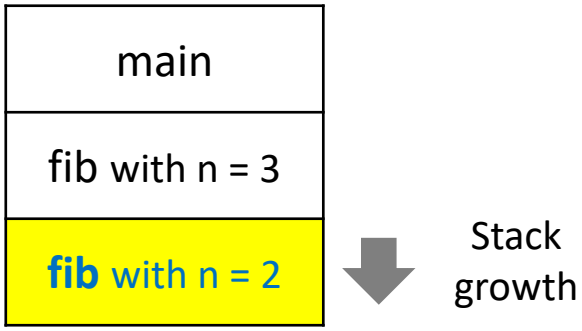
An example of activation trees

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   fib(3);
}
```



fib(2) is called in fib(3)



[stack]

Currently active function: fib with n = 2

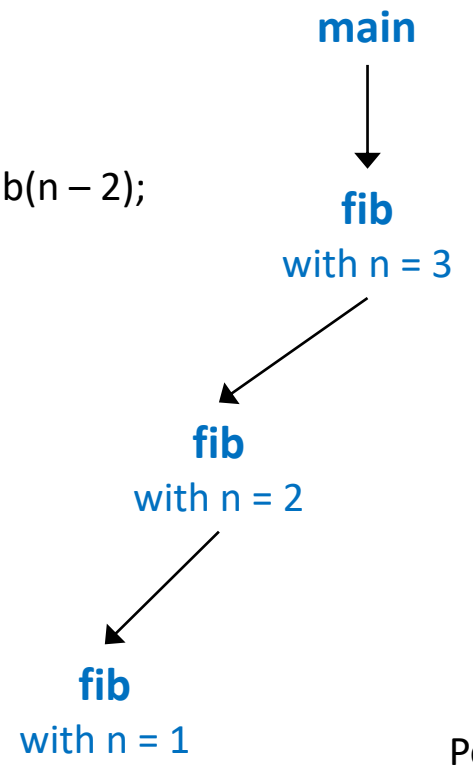
Pending functions: fib with n = 3, main

Activations

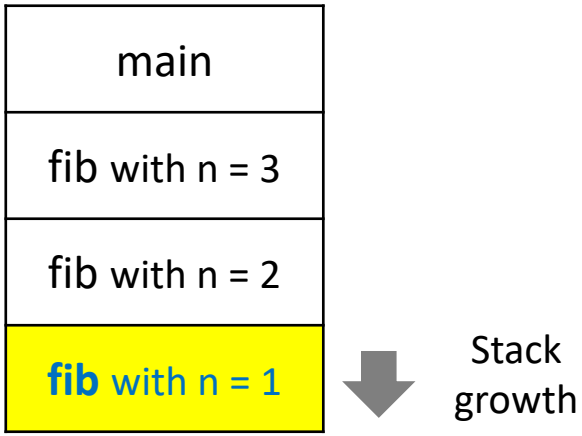
An example of activation trees

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   fib(3);
}
```



fib(1) is called in fib(2)



[stack]

Currently active function: fib with n = 1

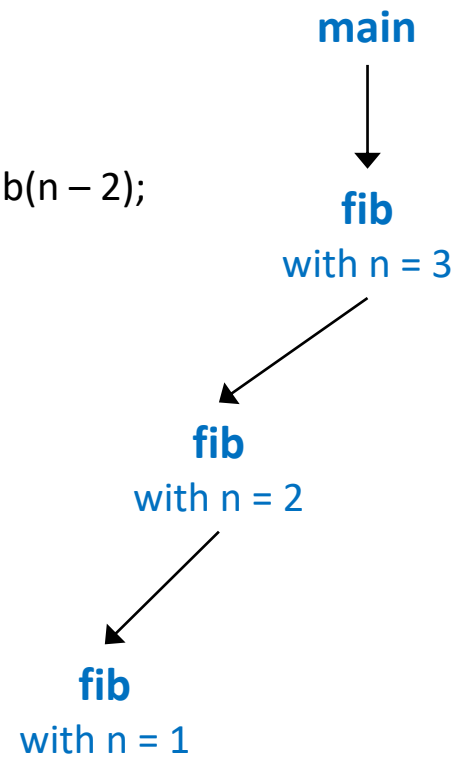
Pending functions: fib with n = 2, fib with n = 3, main

Activations

An example of activation trees

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   fib(3);
}
```



After completing the execution of fib(1)

main
fib with n = 3
fib with n = 2

Popped out!!

fib with n = 1

[stack]

Currently active function: fib with n = 2

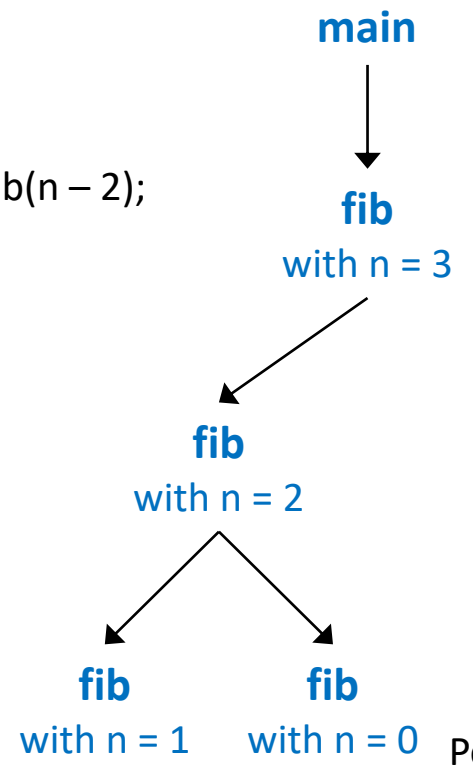
Pending functions: fib with n = 3, main

Activations

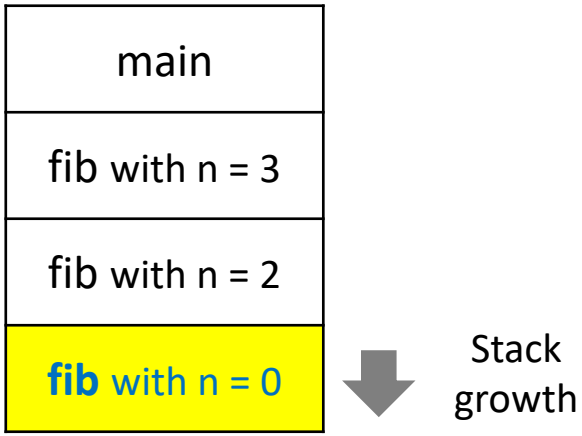
An example of activation trees

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   fib(3);
}
```



fib(0) is called in fib(2)



[stack]

Currently active function: fib with n = 0

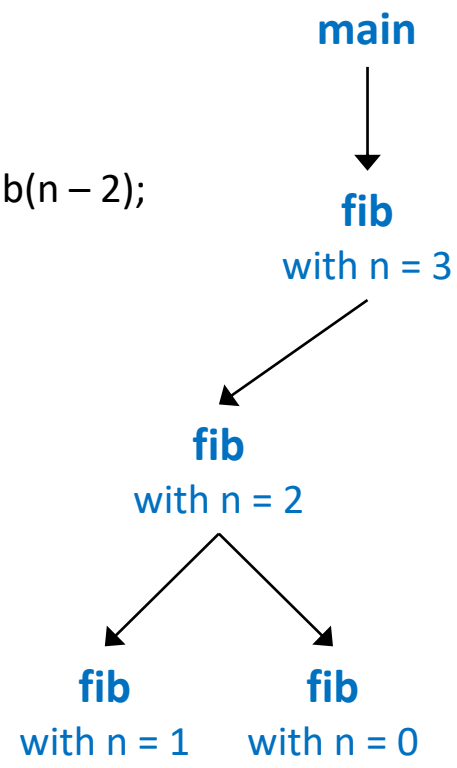
Pending functions: fib with n = 2, fib with n = 3, main

Activations

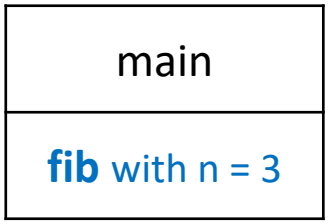
An example of activation trees

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   fib(3);
}
```



After completing the execution of fib(0) and fib(2)



Popped out!!

fib with n = 0

fib with n = 2

[stack]

Currently active function: fib with n = 3

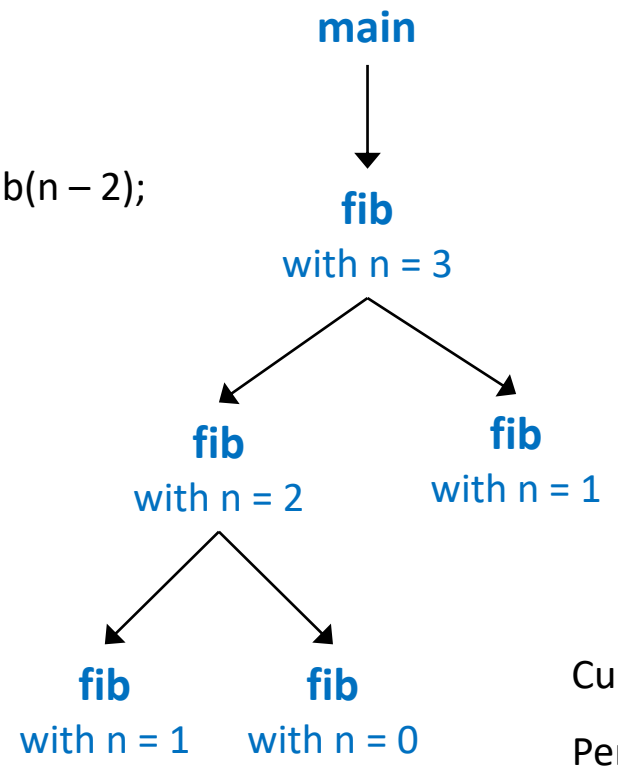
Pending functions: main

Activations

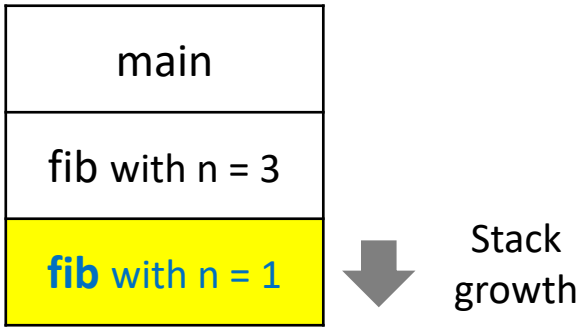
An example of activation trees

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   fib(3);
}
```



fib(1) is called in fib(3)



[stack]

Currently active function: fib with n = 1

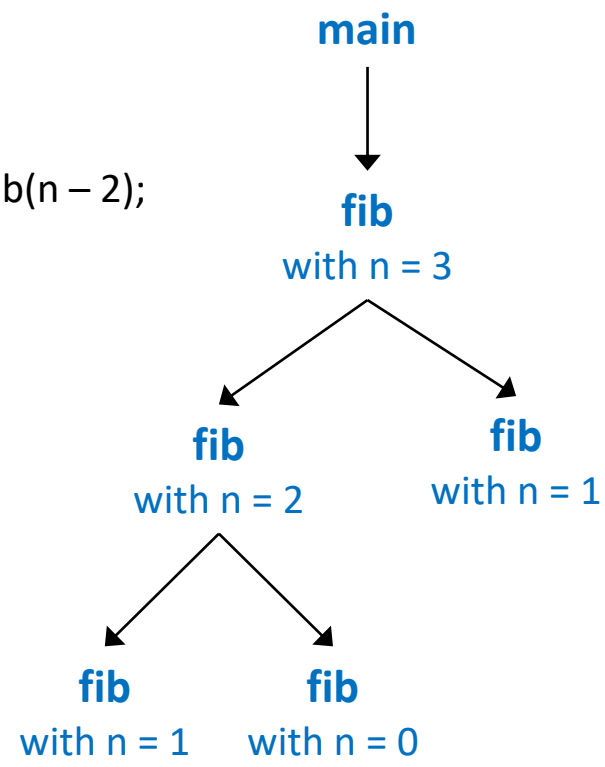
Pending functions: fib with n = 3, main

Activations

An example of activation trees

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   fib(3);
}
```



After completing all the execution

Popped out!!

fib with n = 1

fib with n = 3

main

[stack]

Currently active function: none

Pending functions: none

Function representations

Two main questions related with function representations

1) How to represent the relationship between functions?

(e.g., function calls)

Answer: use a stack!!

2) How to keep the information needed to manage functions?

(e.g., parameters, return values, return address, …)

Activation records

The information needed to manage an activation of function F

- Input parameters

  This is supplied by the caller of F

- Space for F's return value

  This is needed by the caller of F and filled when F is completed

- Control link: a pointer to the previous activation record (the caller of F)

- Machine status prior to calling F (e.g., return address, contents of registers)

- F's local, temporary variables

Activation records

Examples

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n – 1) + fib(n – 2);
}

int main () {
   int a = fib(1);
}
```



```
begin fib
   t0 = n <= 1
   if t0 goto L0

   t1 = n - 1
   param t1
   t2 = call fib, 1

   t3 = n - 2
   param t3
   t4 = call fib, 1

   t5 = t2 + t4
   return t5
L0:
   return n
end fib
```

```
begin main
   param 1
   t0 = call fib, 1
   a = t0
end main
```

Activation records

Examples: an activation record of fib(1)

Space for return value	(result)
Input parameter	n = 1
Control link	A start address of the activation record of main
Saved machine status (e.g., return address)	A memory address of code executed after fib(1) in main
Space for local and temporary variables	t0
	t1
	t2
	t3
	t4
	t5

```
begin fib
    t0 = n <= 1
    if t0 goto L0

    t1 = n - 1
    param t1
    t2 = call fib, 1

    t3 = n - 2
    param t3
    t4 = call fib, 1

    t5 = t2 + t4
    return t5
L0:
    return n
end fib
```

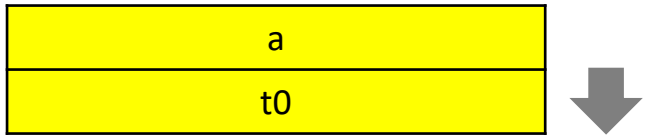
```
begin main
    param 1
    t0 = call fib, 1
    a = t0
end main
```

Activation records

Examples: when main() is called

The activation record of main() is stored into the stack

[stack]




```
begin fib
   t0 = n <= 1
   if t0 goto L0

   t1 = n - 1
   param t1
   t2 = call fib, 1

   t3 = n - 2
   param t3
   t4 = call fib, 1

   t5 = t2 + t4
   return t5
L0:
   return n
end fib
```

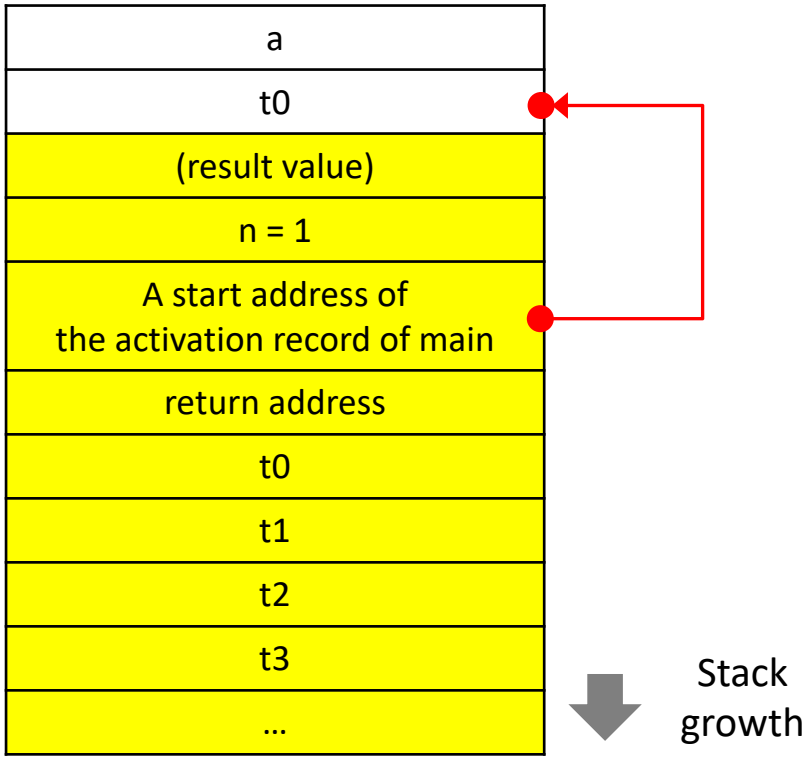
```
begin main
   param 1
   t0 = call fib, 1
   a = t0
end main
```

Activation records

Examples: when fib(1) is called in main()

The activation record of fib(1) is stored into the stack

[stack]




```
begin fib
    t0 = n <= 1
    if t0 goto L0

    t1 = n - 1
    param t1
    t2 = call fib, 1

    t3 = n - 2
    param t3
    t4 = call fib, 1

    t5 = t2 + t4
    return t5
L0:
    return n
end fib
```

```
begin main
    param 1
    t0 = call fib, 1
    a = t0
end main
```

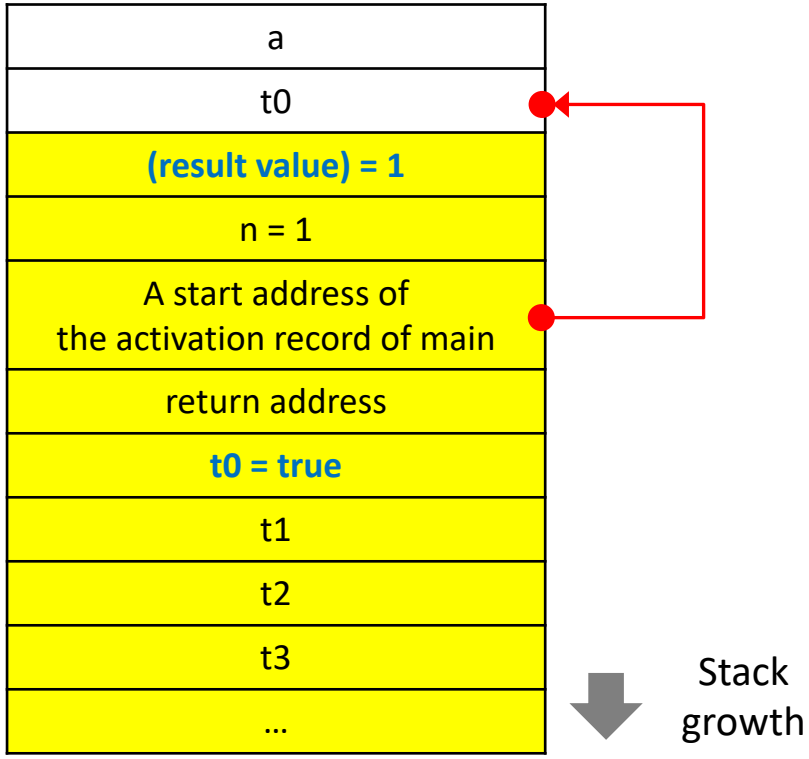
Activation records

Examples: when fib(1) is executed

The program will use the stored input parameter 1

and eventually store the return value 1

[stack]




```
begin fib
    t0 = n <= 1
    if t0 goto L0

    t1 = n - 1
    param t1
    t2 = call fib, 1

    t3 = n - 2
    param t3
    t4 = call fib, 1

    t5 = t2 + t4
    return t5
L0:
    return n
end fib
```

```
begin main
    param 1
    t0 = call fib, 1
    a = t0
end main
```

Activation records

Examples: after fib(1) is executed

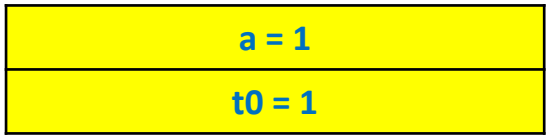
All the contents of the activation record of fib(1) is popped out

1) the program returns back to the return address and

2) do remaining works

(e.g., the return value is copied to t0, …)

[stack]




```
begin fib
   t0 = n <= 1
   if t0 goto L0

   t1 = n - 1
   param t1
   t2 = call fib, 1

   t3 = n - 2
   param t3
   t4 = call fib, 1

   t5 = t2 + t4
   return t5
L0:
   return n
end fib
```

```
begin main
   param 1
   t0 = call fib, 1
   a = t0
end main
```

Function representations

Two main questions related with function representations

1) How to represent the relationship between functions?

(e.g., function calls)

Answer: use a stack!!

2) How to keep the information needed to manage functions?

(e.g., parameters, return values, return address, …)

Answer: store an activation record

Runtime environment

A set of data structures used at runtime to support high-level structures

This environment deals with a variety of issues

- **What do objects look like in memory?**

  The representation of the objects in memory space

- **What do functions look like in memory?**

  The linkages between functions

  The mechanisms passing parameters

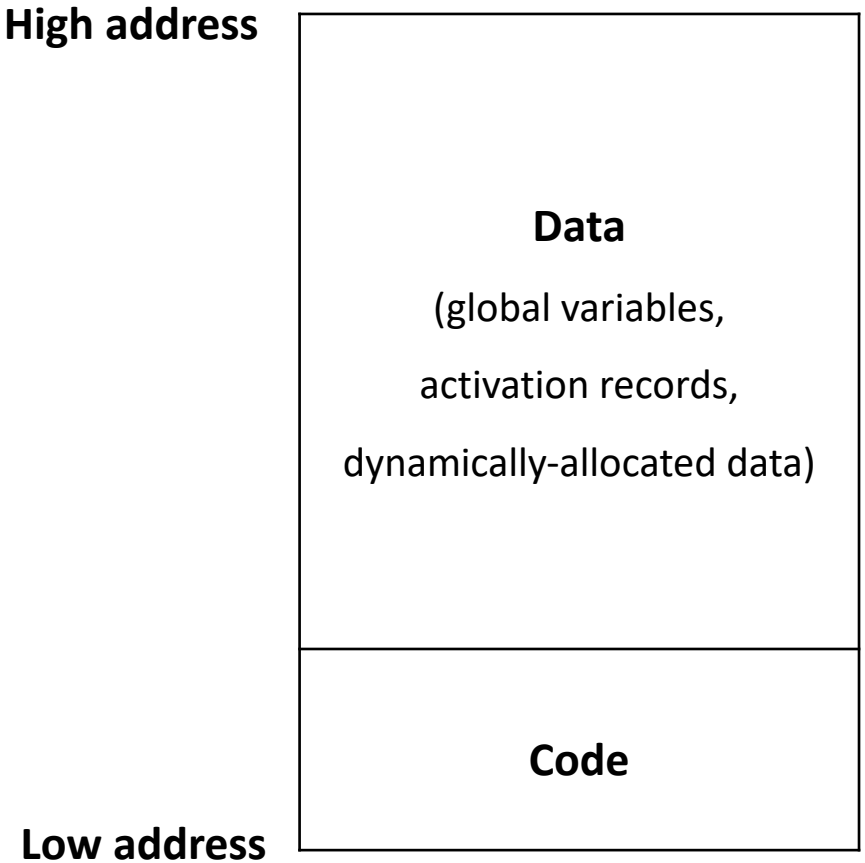
- **Where in memory should objects and functions be placed?**

  The layout and allocation of memory space for the objects and functions

Memory layout

Compilers determine how code and data are stored in memory

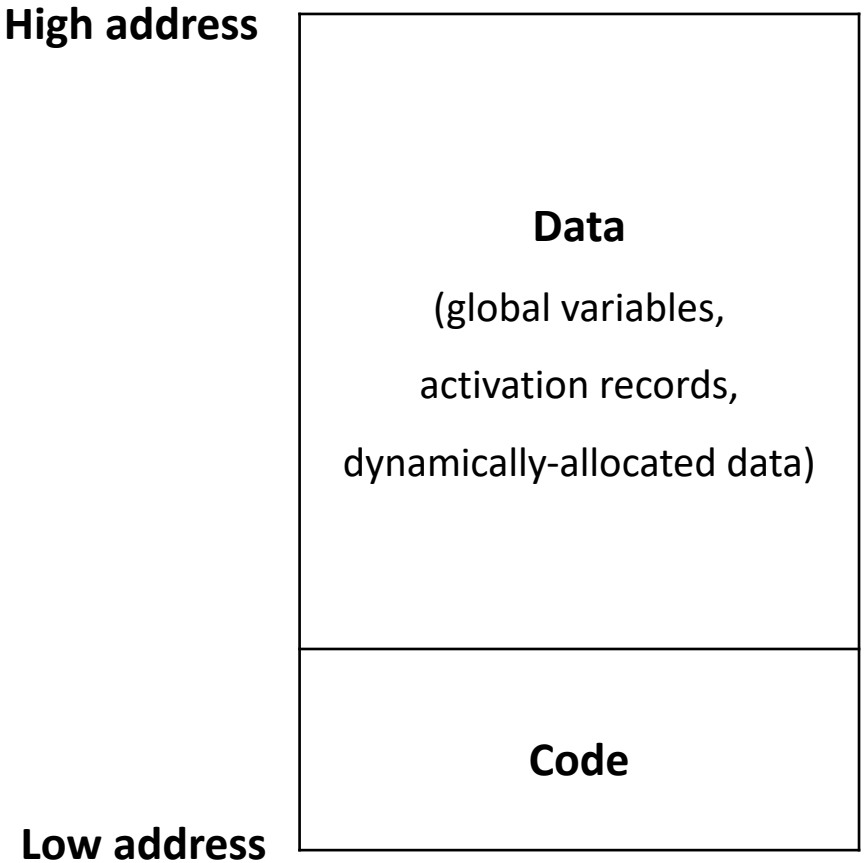
Assumption: a program uses contiguous memory space



Memory layout

Compilers determine how code and data are stored in memory

Assumption: a program uses contiguous memory space



Global variables

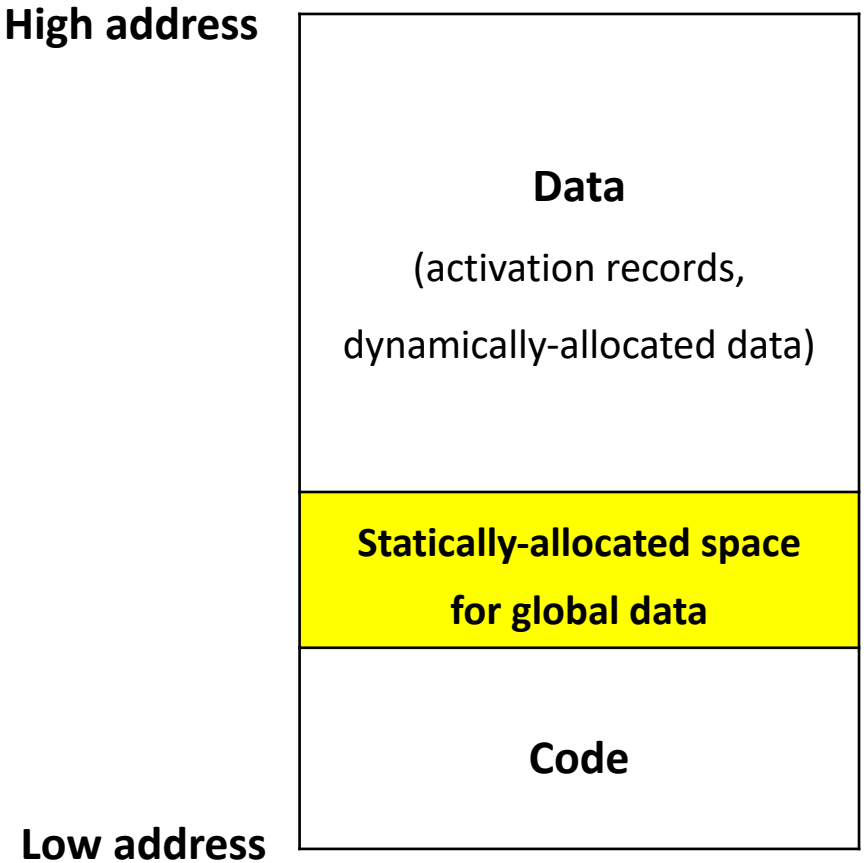
- All references to a global variable point to the same object
- Global variables cannot be stored in an activation record
- **Instead, global variables are assigned a fixed address**

  statically allocated!!

Memory layout

Compilers determine how code and data are stored in memory

Assumption: a program uses contiguous memory space



Global variables

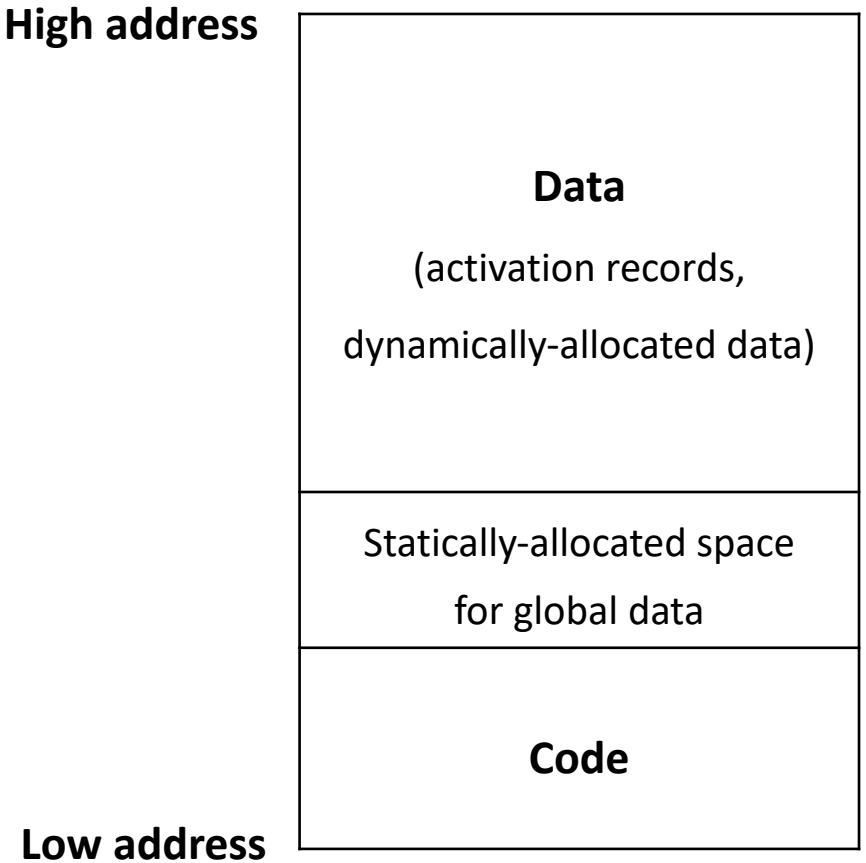
- All references to a global variable point to the same object
- Global variables cannot be stored in an activation record
- **Instead, global variables are assigned a fixed address**

  statically allocated!!

Memory layout

Compilers determine how code and data are stored in memory

Assumption: a program uses contiguous memory space



Activation records

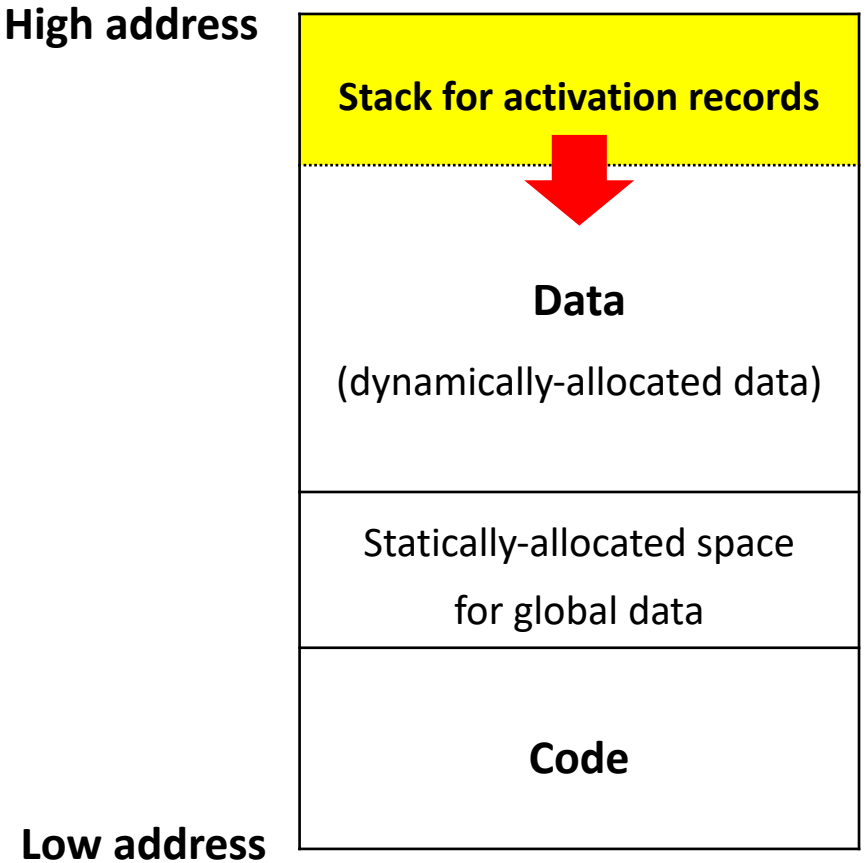
- Managed in a stack
- **The stack grows**

  from high addresses to low addresses

Memory layout

Compilers determine how code and data are stored in memory

Assumption: a program uses contiguous memory space



Activation records

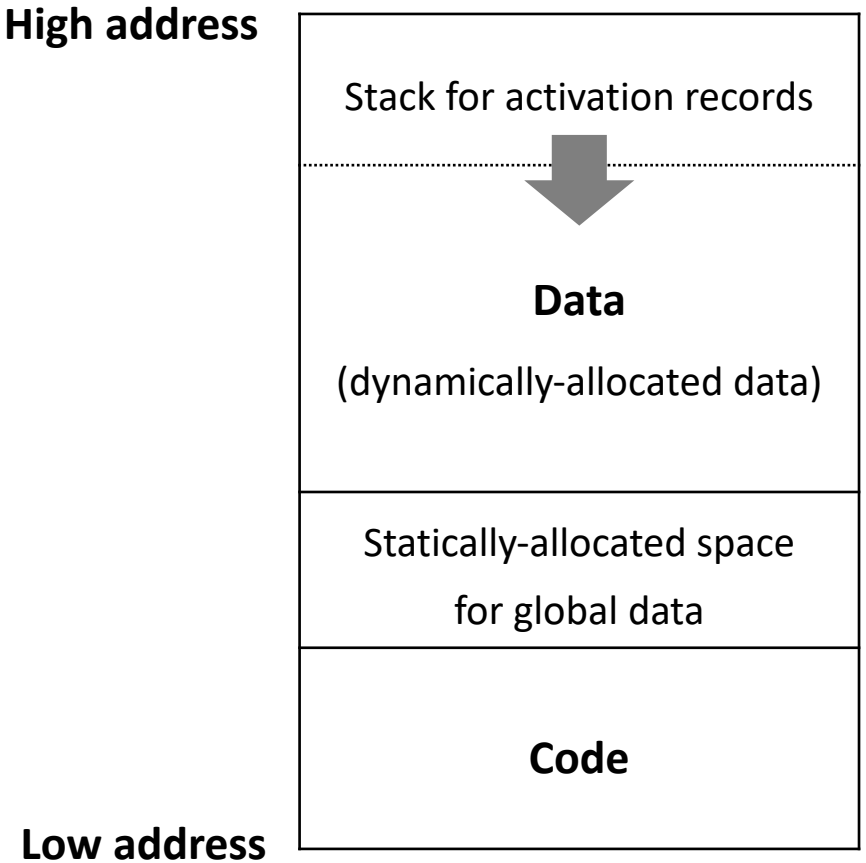
- Managed in a stack
- **The stack grows**

  from high addresses to low addresses

Memory layout

Compilers determine how code and data are stored in memory

Assumption: a program uses contiguous memory space



Dynamically-allocated data

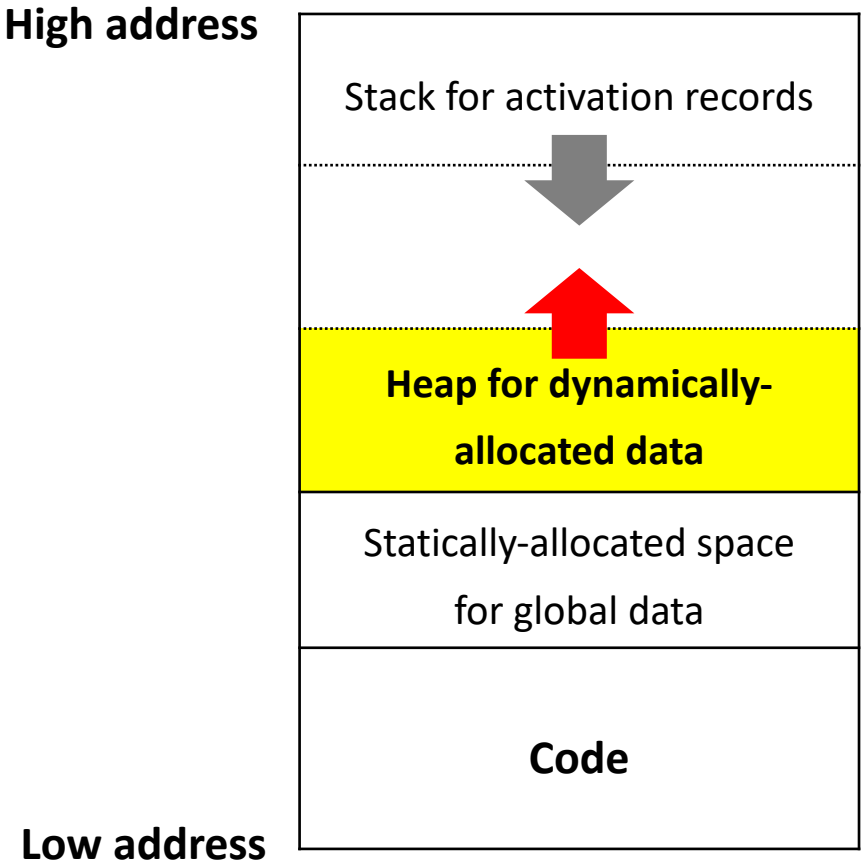
Examples in C

- int* a = (int*)malloc(sizeof(int) * 10);
- a's memory space is dynamically allocated at runtime
- When free(a) is called,
  a's memory space is dynamically deallocated

Memory layout

Compilers determine how code and data are stored in memory

Assumption: a program uses contiguous memory space



Dynamically-allocated data

- **Dynamically-allocated data is managed in a heap**
- **The heap grows from low addresses to high addresses**

  "To avoid the overlap between the stack and heap"

Summary: runtime environment

A set of data structures used at runtime to support high-level structures

This environment deals with a variety of issues

- **What do objects look like in memory?**

  ”Data structure alignment, array representations…”
- **What do functions look like in memory?**

  “Keep activation records of functions in a stack”
- **Where in memory should objects and functions be placed?**

  “Determine memory layout for code and data”

“Compiler determines, at compile time, the memory layout of code and data, and generates code that correctly accesses the location of target data”