DIENS: Semaine Informatique Pratique
Programming with C

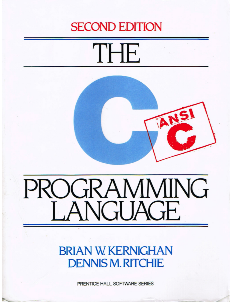
T. Bourke

September 2023

The C language

- Originally designed for and implemented on the UNIX operating system on the DEC PDP-11 by D. Ritchie.
  BCPL (M. Richards, UK, 1967) $\rightarrow$ B (K. Thompson & D. Ritchie, USA, 1969) $\rightarrow$ C (1972)
  (Basic Combined Programming Language)

- Designed for portable systems programming; very widely used.



1978	K&R	Informal standard
1989	C89/C90/ANSI C	International standard
1999	C99	Several new features
2011	C11	Several new features
2017	C17	Correct defects in C11
2023?	C2x	Informal name

Why C?

Assembly language

- Program a specific architecture directly.
- Explicitly specify instructions and registers.
- Manually express control structures: branching, loops, stack.
- Manually express data structure layout using bits, bytes and words.
- Some convenience: mnemonics, labels, macros.

Java, OCaml, etc.

- Abstract notion of machine (e.g., reduce λ-terms with side-effects)
- Memory safety: strong typing, garbage collection, bounds checking
- Advanced abstractions: objects, function closures, exceptions

Why C?

C

- Program an abstract machine directly and compile for different architectures.
- The compiler takes care of
  - register allocation and use,
  - implementing control structures and the stack, and
  - laying out data structures.
- But the programmer maintains a lot of control over exactly how the program executes and manipulates memory.
- A ‘sharp knife’ (easy to lose a finger).

Modern trends

- Static analysis: from `lint` and compiler warnings to Astrée
- Bounds declaration and checking: see microsoft/checkedc

The C standard

Undefined behavior

No requirements on nonportable or erroneous constructs, e.g., C11 §6.5.5,

> *The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.*

- Possible behaviours for division by zero:
  - The compiler rejects the program—not always possible.
  - The program is terminated—typical.
  - The program keeps running with an arbitrary value as a result—allowed.
- Facilitates portability: less constraints on compilers and hardware.
- Permits optimisation for speed and memory use:
  - Runtime checks are not obligatory.
  - Compilers may optimize for defined behaviours.
- See: Regehr, A Guide to Undefined Behavior in C and C++.

The C standard

Unspecified behavior

The standard allows more than one possibility, e.g., order of evaluation for function arguments, C11 §6.5.2.2 (10)

> *There is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call. Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.*

Implementation-defined behavior

- The compiler documentation must state how an unspecified behavior is resolved (on a specific architecture or platform).
- See, e.g., the documentation for gcc, CompCert, or Microsoft C.

The C standard

The C standard is written in English. It is precise but requires interpretation.

- Different people (compiler writers, programmers) may make different interpretations.
- How to interpret tricky corner cases?
- Difficult to use as a base for precise reasoning, i.e., static analysis, verification.
- Important systems are written in C...

Can we, computer scientists & engineers, do better?

The C standard

The C standard is written in English. It is precise but requires interpretation.

- Different people (compiler writers, programmers) may make different interpretations.
- How to interpret tricky corner cases?
- Difficult to use as a base for precise reasoning, i.e., static analysis, verification.
- Important systems are written in C...

Can we, computer scientists & engineers, do better?

- Yes: significant subsets of the C standard have been formalized in logic.
- Computer assistance is a practical necessity.
- Formalization in Higher-Order Logic: [Norrish (1998): C formalised in HOL]
- Formalization in Coq: [Blazy and Leroy (2009): Mechanized Semantics for the Clight Subset of the C Language]

Plan

What is a C program?

A text file containing a sequence of declarations and definitions of

- types (`struct`/`union`/`enum`) and type aliases (`typedef`)
- static variables (visible globally or within a 'module')
- functions

Compile it to an executable file:

- `cc helloworld.c; ./a.out`
- `cc -Wall -o helloworld helloworld.c`

```c
// helloworld.c
int c = 0;

int puts(const char *s);
void inc(void);

int main(void)
{
  inc();
  inc();
  puts("hello world");

  return c; // echo $?
}

void inc(void)
{
  c = c + 1;
}
```

Importing libraries

External libraries have two parts:

- **interface**: a sequence of declarations
- **implementation**: the compiled definitions

The interface is declared in a *header file* whose contents are copied into the program by the *preprocessor*.

- Looks for Header files in the `search` path
- `#include "mylib.h"`: your files
- `#include <stdio.h>`: system files
- `cpp -I/custom/path -v`
  `clang -x c -v /dev/null`

```c
// helloworld2.c

#include <stdio.h>

int main(void)
{
  puts("hello world");
  return 0;
}
```

See the results of preprocessing:
`cpp -E helloworld2.c`

What's in a C function definition?

- Return type, name, list of arguments (types and names)
  - `int main(void)`
  - `int main(int argc, char* argv[])`
  - `int main(int argc, char* argv[], char *environ[])`
- Variable declarations and initializations: `int c = 0;`

- Statements
  - assignment
    (`x = e`)
  - selection
    (`if`/`else`, `switch`)
  - iteration
    (`for`, `while`, `do while`)
  - jumps
    (`goto`, `continue`, `break`, `return`)

- Expressions
  - machine operations:
    arithmetic, logic, relational
  - variables
  - literal ('concrete' constants)
  - function calls
  - with side effects
    (increment, decrement, assignment)
  - type casts: `(float)c`
  - `sizeof` expression or type
  - conditional (`e ? e : e`)

Constants

- Literal values:
  e.g., 7, `"hello world"`
- `const` keyword:
  `const int dimensions = 3;`
- Preprocessor definitions:
  `#define DIMENSIONS 7`
  - Source file = sequence of *preprocessor tokens* and white space
  - A defined token is blindly replaced by its definition.

Constants

- Literal values:
  e.g., 7, `"hello world"`
- `const` keyword:
  `const int dimensions = 3;`
- Preprocessor definitions:
  `#define DIMENSIONS 7`
  - Source file = sequence of *preprocessor tokens* and white space
  - A defined token is blindly replaced by its definition.
- Preprocessor macros:

```c
//preproc.c

#define SQR(x) x*x

int a = 3;

int main(void)
{
    return SQR(a + 1);
}
```

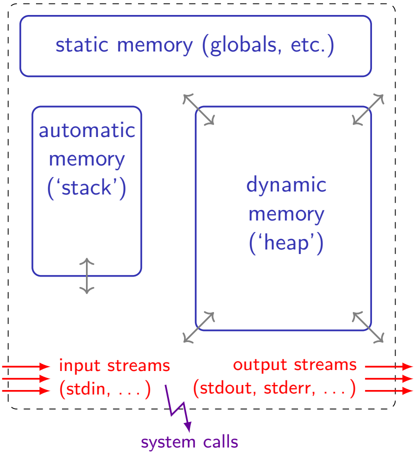
Constants

- Literal values:
  e.g., 7, `"hello world"`
- `const` keyword:
  `const int dimensions = 3;`
- Preprocessor definitions:
  `#define DIMENSIONS 7`
  - Source file = sequence of *preprocessor tokens* and white space
  - A defined token is blindly replaced by its definition.
- Preprocessor macros:

```c
//preproc.c

#define SQR(x) x*x

int a = 3;

int main(void)
{
    return SQR(a + 1);
}
```

- Better: `#define SQR(x) ((x)*(x))`

Programming model



- Statements/expressions execute on an *abstract machine*, which transitions from state to state.
- **Static memory**: fixed-size; variable lifetime = program lifetime
- **Automatic memory**: grows and shrinks with function calls; variable lifetime = function lifetime
- **Dynamic memory**: grows and shrinks as required; variable lifetime = manually managed by the programmer
- Programs interact with the system by
  - reading streams of bytes
  - writing streams of bytes
  - making special function calls

Plan

Basic arithmetic types

`char`	`'a'`, `'0'`, `'\n'`
	`'\''`
	`'\060'`, `'\xff'`
		byte/single character

`int`	`-1`
	`18`
	`022`
	`0x12`
		signed integer type
		that is ‘natural’ for architecture
		(not necessarily a machine word)

`unsigned int`	`18U`, `022U`, `0x12U`	unsigned integer type

`double`	`-1.`
	`3.141`
	`12.3e-2`
		(double-precision) floating-point type

Basic arithmetic types: storage

- The number of bytes required to represent a value varies by type and depends on the target platform (architecture and OS). The standard only guarantees minimum sizes.
- The `sizeof` operator returns the size in bytes of a type or expression.
- `limits.h` defines constants for the ranges of integer types.
- `float.h` defines constants for the characteristics of floating-point types.

```c
#include <stdio.h>
#include <limits.h>
#include <float.h>

int main()
{
  printf("sizeof(char)=%zu [%hhd, %hhd]\n", sizeof(char), CHAR_MIN, CHAR_MAX);
  printf("sizeof(int)=%zu [%d, %d]\n", sizeof(int), INT_MIN, INT_MAX);
  printf("sizeof(unsigned int)=%zu [0, %u]\n", sizeof(unsigned int), UINT_MAX);
  printf("sizeof(double)=%zu [%g, %g]\n", sizeof(double), -FLT_MAX, FLT_MAX);

  return 0;
}
```

Other arithmetic types

`_Bool`	boolean type (0, 1)
`bool`	with `<stdbool.h>` (`false`, `true`)
`signed char`	Is `char` signed or unsigned?
`unsigned char`	—it's implementation-dependent
`short (int)`	use less memory
`long (int)`	bigger integer range
`long long (int)`	often the same as `long`
`unsigned short (int)`	use less memory
`unsigned long (int)`	bigger integer range
`unsigned long long (int)`	often the same as `unsigned long`
`float`	smaller with less precision than `double`
`long double`	same or more precise than `double`
`double/float _Complex`	pair of floating-point numbers

Exact-width integer types: stdint.h

- `int8_t`, `int16_t`, `int32_t`, `int64_t`
- `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

- Useful for file formats and network communications.
- May be preferred to `char`, `short`, and `long` in some projects and style guides.

Advice

- Write concise, direct programs without worrying much about low-level performance and memory size.
- Tweak later only if necessary based on empirical data.

Arithmetic

- Operator meaning depends on argument types: `+`, `-`, `*`, `/`, `%`
- E.g, `x + y`:
  - 32-bit integer addition
  - 64-bit integer addition
  - single-precision floating-point addition
  - double-precision floating-point addition
- Unsigned integer operations never overflow, they are calculated modulo 2^n.
- Signed integer overflow is undefined, though most compilers provide control:
  - `-fwrapv`: on overflow, wrap-around in 2s complement.
  - `-ftrapv`: on overflow, generate a trap (may add explicit checks).

```c
// cc -c -S add.c      see also https://godbolt.org (without Intel ASM syntax)

int iplus(int x, int y)
{
    return x + y;
    // addl %edx, %eax
    // l = 32-bits
}

char cplus(char x, char y)
{
    return x + y;
    // addl %edx, %eax
    // l = 32-bit words
}

long lplus(long x, long y)
{
    return x + y;
    // addq %rdx, %rax
    // q = 64-bit words
}

float fplus(float x, float y)
{
    return x + y;
    // addss -8(%rbp), %xmm0
    // Scalar Single-Precision FP
}

double dplus(double x, double y)
{
    return x + y;
    // addsd -16(%rbp), %xmm0
    // Scalar Double-Precision FP
}
```

Usual arithmetic conversions

- (+) has type "signed char $\to$ signed char $\to$ int".
- Why?

Usual arithmetic conversions

- (+) has type "signed char → signed char → int".
- Why?

- Even though the arguments may each be stored in one byte of memory, the result is calculated using registers and an Arithmetic Logic Unit.
- The standard C11 §6.3 expresses this idea using *implicit type conversions*, *integer promotions*, and *usual arithmetic conversions*.
- The basic idea:
  - » Convert integer arguments upward to `int` or `unsigned int`.
  - » Convert both arguments to a common 'larger' type preserving sign and value where possible
    E.g., "$(+)$: int → double → double".

- Expressed precisely and concisely in CompCert: cfrontend/Cop.v

```
Inductive binarith_cases: Type :=
  | bin_case_i (s: signedness)        (**r at int type *)
  | bin_case_l (s: signedness)        (**r at long int type *)
  | bin_case_f                        (**r at double float type *)
  | bin_case_s                        (**r at single float type *)
  | bin_default.                      (**r error *)

Definition classify_binarith (ty1: type) (ty2: type) : binarith_cases :=
 match ty1, ty2 with
  | Tint I32 Unsigned _     , Tint _ _ _              ⇒ bin_case_i Unsigned
  | Tint _ _ _              , Tint I32 Unsigned _     ⇒ bin_case_i Unsigned
  | Tint _ _ _              , Tint _ _ _              ⇒ bin_case_i Signed
  | Tlong Signed _          , Tlong Signed _          ⇒ bin_case_l Signed
  | Tlong _ _               , Tlong _ _               ⇒ bin_case_l Unsigned
  | Tlong sg _              , Tint _ _ _              ⇒ bin_case_l sg
  | Tint _ _ _              , Tlong sg _              ⇒ bin_case_l sg
  | Tfloat F32 _            , Tfloat F32 _            ⇒ bin_case_s
  | Tfloat _ _              , Tfloat _ _              ⇒ bin_case_f
  | Tfloat F64 _            , (Tint _ _ _ | Tlong _ _) ⇒ bin_case_f
  | (Tint _ _ _ | Tlong _ _), Tfloat F64 _            ⇒ bin_case_f
  | Tfloat F32 _            , (Tint _ _ _ | Tlong _ _) ⇒ bin_case_s
  | (Tint _ _ _ | Tlong _ _), Tfloat F32 _            ⇒ bin_case_s
  | _                       , _                       ⇒ bin_default
```

Explicit type conversion

Convert a value from one type to another (NB: conversion != cast), e.g.,

```
#include <stdio.h>

int main()
{
    signed char w = -1; // movb: move byte
    unsigned int x = (unsigned int)w;
    // movsbl: move byte to 'long' with sign extend (no need for cast)

    unsigned int y = (unsigned char)w;
    // movzbl: move byte to 'long' with zero extend

    double z = (double)w;
    // cvtsi2sd: convert integer to double-precision float (no need for cast)

    printf("w=0x%hhx (%hhd) x=0x%08x (%u) y=0x%08x (%u) z=%g\n",
           w, w, x, x, y, y, z);

    return 0;
}
```

Type conversion to _Bool

```
#include <stdio.h>
#include <stdbool.h>

int main()
{
    int x = 0;
    int y = 42;
    int xb = (bool)x;
    int yb = (bool)y;

    printf("xb=%d yb=%d\n", xb, yb);

    return 0;
}
```

Type conversion to _Bool

```
#include <stdio.h>
#include <stdbool.h>

int main()
{
    int x = 0;
    int y = 42;
    int xb = (bool)x;
    int yb = (bool)y;

    printf("xb=%d yb=%d\n", xb, yb);

    return 0;
}
```

- 0 remains 0, meaning `false`.
- Any other integer becomes 1, meaning `true`.

Operators: logical versus bitwise

```c
#include <stdio.h>

int main()
{
    unsigned int x = 0x6; // 0b1010
    unsigned int y = 0x5; // 0b1001

    printf("x & y = 0x%x; x && y = 0x%x\n", x & y, x && y);
    printf("x | y = 0x%x; x || y = 0x%x\n", x | y, x || y);
    printf(" ~x = 0x%x; !x = 0x%x\n", ~x, !x);
}
```

- Other bitwise operators
  - `x ^ y`: exclusive-or (xor)
  - `x << 3`: left-shift, fills with zeros
  - `x >> 3`: right-shift $\begin{cases} \text{fills with zeros if unsigned or positive,} \\ \text{otherwise implementation-defined, usually using the sign bit} \end{cases}$
- The logical `&&` and `||` have short-circuit behaviour:
  the expression at right is only evaluated if necessary.

Operators with side-effects

Expressions in C may have side-effects.

Important example: prefix increment and postfix increment operators.

```
int x = 0;
int y = x++ + 3;
int z = ++x + 3;
```

Similar program in OCaml

```
let x = ref 0;
let y = (let r = !x in x := !x + 1; r) + 3;;
let z = (x := !x + 1; !x) + 3;;
```

- Also `--x`, `x--`, `x += y` (`x = x + y`), `x <<= 2` (`x = x << 2`)
- Important: the equality operator is `==` (inequality is written `!=`)
- The assignment operator `=` updates a variable and returns the value.
  Common mistake: `if (x = 3) { ... };`

Plan

Control structures: selection with if/else/else-if

```
if (x > 0) {
  printf("x is positive\n");
} else if (x < 0) {
  printf("x is negative\n");
} else {
  printf("x is zero\n");
}
```

- Braces are optional but usually a good idea.
  - `;` terminates a statement or declaration, but not a {compound} statement.
  - `else` is bound to innermost `if`
- The first branch is taken if the expression does not evaluate to zero.
- The `else` branch is optional.

Control structures: selection with switch

```c
switch (c) {
  case 'A':
    printf("c is 'A'\n");
    break;

  case 'B':
    printf("c is 'B'\n");

  case 'C':
    printf("c is 'B' or 'C'\n");
    break;

  default:
    printf("c is not 'A', 'B', or 'C'\n");
}
```

- Evaluate the expression and jump to the first label that matches.
- Jump to the `default` branch if no label matches, or if there is none, to the statement after the `switch`.
- Keep executing statements until a `break` or the closing `}`.
- This is known as *fall-through execution*.

Control structures: looping with while

```c
int i;

i = 0;
while (i >= 0) {
  printf("countdown: %d\n", i);
  --i;
}
```

1. Evaluate the guard expression.
2. If not 0, then execute the loop body and repeat, otherwise continue after the loop.

- `break`: early exit from innermost loop.
- `continue`: end iteration, reevaluate guard.

Control structures: looping with for

```c
int i, j;

// init; guard; after
for (i = 10; i >= 0; --i) {
  printf("countdown: %d\n", i);
}

for (i = 10, j = 0; i >= 0; --i, j++) {
  printf("down: %d up: %d\n", i, j);
}
```

1. Evaluate the *init* expression.
2. Evaluate the *guard* expression.
3. If 0 then continue after the loop.
4. Otherwise execute the loop body.
5. Evaluate the *after* expression and repeat from step 2.

- `break`: early exit from innermost loop.
- `continue`: skips to step 5.
- Comma operator: a sequence of expressions, taking the last value.

Functions

```c
// declaration
long factorial(int n);

int main(void)
{
  return factorial(7);
}

// definition (and declaration)
long factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

- Distinguish function *declarations* from *definitions*.
- A declaration, or function prototype, provides the function name, argument types, and return type.
- This information suffices to use the function; it allows type checking and code generation for calls.
- A definition must be given somewhere and must agree with any earlier declarations.
- A definition also counts as a declaration.
- A `return` statement returns control, and usually a result, to the caller.

Plan

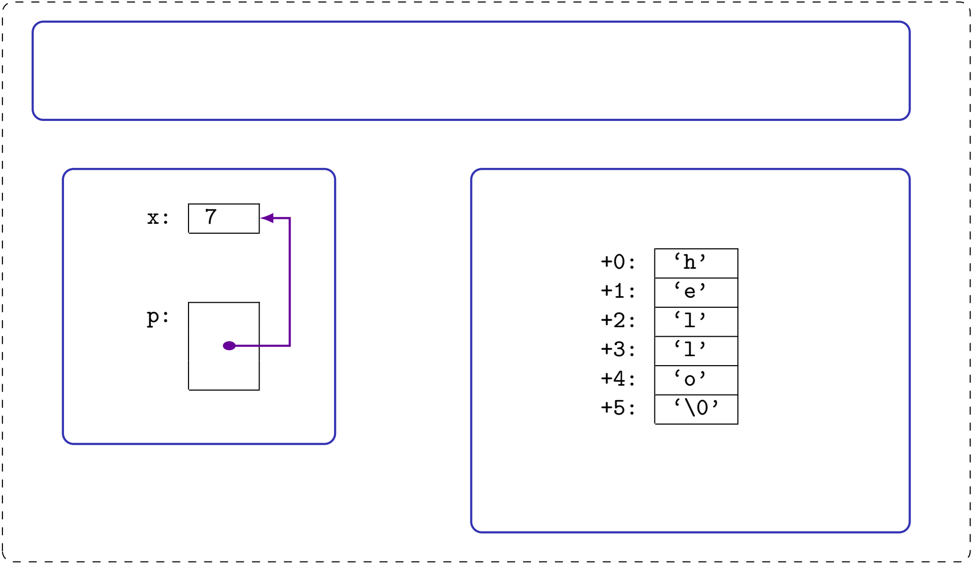
Memory model: machine

0x0000···0000:	0x??
0x0000···0001:	0x??
0x0000···0002:	0x??
⋮	⋮
0x0a20···0008:	0x07
0x0a20···0009:	0x??
0x0a20···000a:	0x??
0x0a20···000b:	0x??
⋮	⋮
0x0a20···0010:	0x08
0x0a20···0011:	0x00
0x0a20···0012:	0xee
0x0a20···0013:	0x00
⋮	⋮
0xFFFF···FFFD:	0x??
0xFFFF···FFFE:	0x??
0xFFFF···FFFF:	0x??

- A big table of bytes, each with an address
- Distinct and contiguous static, automatic (stack), and dynamic (heap) areas of the table.

```c
// pointer1.c
#include <stddef.h>

int main(void)
{
  char x = 0x07; // allocated on stack
  char *p = NULL; // also allocated on stack...

  p = &x; // address-of a variable
  *p = *p + 3; // dereference a pointer
  p = p + 3; // increment a pointer

  return x; // echo $?
}
```

Memory model: C



Memory model: C (cont.)

- Memory: a set of (isolated) blocks, each indexed locally
- CompCert: pointers are represented by block/offset pairs: (b, i) : block $\times$ nat
  each block b is an array of bytes indexed by $0 \leq i < \mathit{blocksize}$.
- Can only compare / subtract pointers (b_1, i_1) and (b_2, i_2) if $b_1 = b_2$.
  `ptrdiff_t d = p1 - p2`
- `int *q = (int *)p + (int)n`:
  - if $p = (b, i)$ then $q = (b, i + n \cdot \texttt{sizeof}(\texttt{int}))$
- May only dereference a pointer (b, i) if $i + n \cdot \texttt{sizeof}(\texttt{int})$ is within the block
- May compare pointers ‘one past the last element’.

Memory model: C (cont.)

- Memory: a set of (isolated) blocks, each indexed locally
- CompCert: pointers are represented by block/offset pairs: (b, i) : block $\times$ nat
  each block b is an array of bytes indexed by $0 \leq i < \mathit{blocksize}$.
- Can only compare / subtract pointers (b_1, i_1) and (b_2, i_2) if $b_1 = b_2$.
  `ptrdiff_t d = p1 - p2`
- `int *q = (int *)p + (int)n`:
  - » if $p = (b, i)$ then $q = (b, i + n \cdot \texttt{sizeof}(\texttt{int}))$
- May only dereference a pointer (b, i) if $i + n \cdot \texttt{sizeof}(\texttt{int})$ is within the block
- May compare pointers 'one past the last element'.

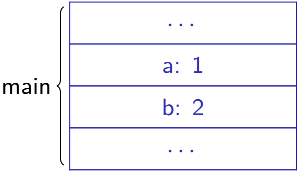
C99 §6.5.9: "Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space."

```c
void swap(int x, int y);

int main()
{
  int a = 1;
  int b = 2;

  swap(a, b);

  return b;
}

void swap(int x, int y)
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

What is wrong with this program?

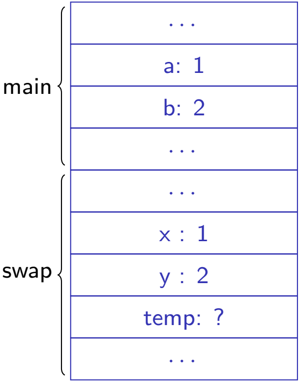
```c
void swap(int x, int y);

int main()
{
  int a = 1;
  int b = 2;

  swap(a, b);

  return b;
}

void swap(int x, int y)
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

What is wrong with this program?



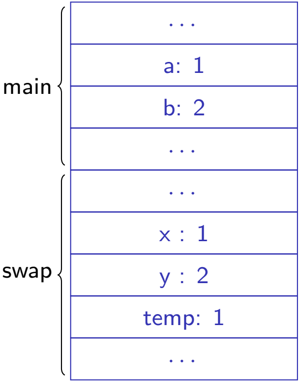
```c
void swap(int x, int y);

int main()
{
  int a = 1;
  int b = 2;

  swap(a, b);

  return b;
}

void swap(int x, int y)
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

What is wrong with this program?



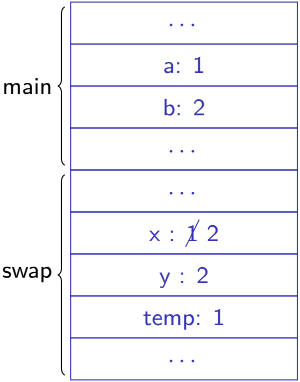
```c
void swap(int x, int y);

int main()
{
  int a = 1;
  int b = 2;

  swap(a, b);

  return b;
}

void swap(int x, int y)
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

What is wrong with this program?



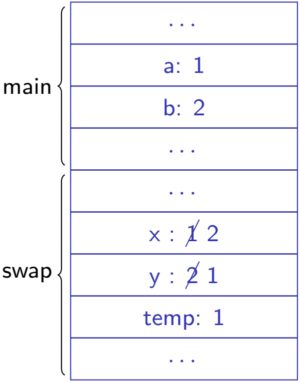
```
void swap(int x, int y);

int main()
{
  int a = 1;
  int b = 2;

  swap(a, b);

  return b;
}

void swap(int x, int y)
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

What is wrong with this program?



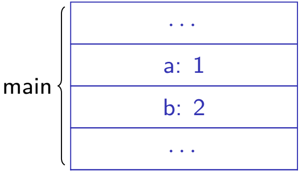
```c
void swap(int x, int y);

int main()
{
  int a = 1;
  int b = 2;

  swap(a, b);

  return b;
}

void swap(int x, int y)
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

What is wrong with this program?



```c
void swap(int x, int y);

int main()
{
  int a = 1;
  int b = 2;

  swap(a, b);

  return b;
}

void swap(int x, int y)
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

What is wrong with this program?

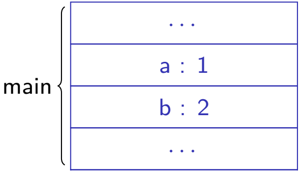


A correct version using pointers.

```c
void swap(int *x, int *y);

int main()
{
  int a = 1;
  int b = 2;

  swap(&a, &b);

  return b;
}

void swap(int *px, int *py)
{
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

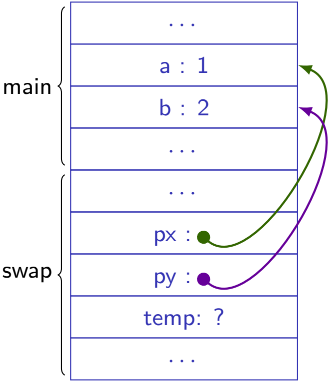
```c
void swap(int *x, int *y);

int main()
{
  int a = 1;
  int b = 2;

  swap(&a, &b);

  return b;
}

void swap(int *px, int *py)
{
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

A correct version using pointers.



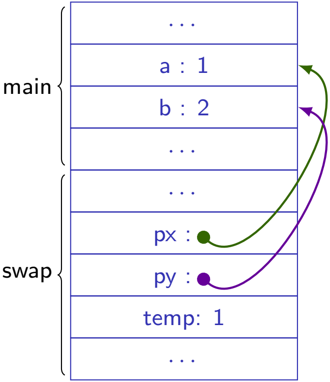
```c
void swap(int *x, int *y);

int main()
{
  int a = 1;
  int b = 2;

  swap(&a, &b);

  return b;
}

void swap(int *px, int *py)
{
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

A correct version using pointers.



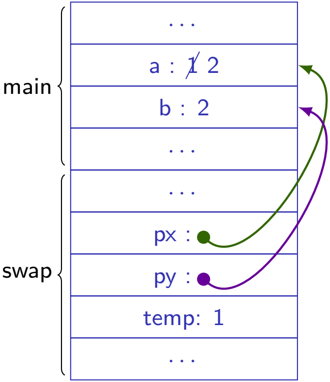
```c
void swap(int *x, int *y);

int main()
{
  int a = 1;
  int b = 2;

  swap(&a, &b);

  return b;
}

void swap(int *px, int *py)
{
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

A correct version using pointers.



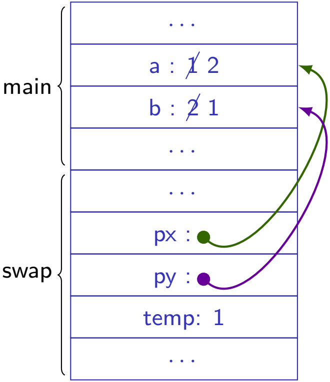
```
void swap(int *x, int *y);

int main()
{
  int a = 1;
  int b = 2;

  swap(&a, &b);

  return b;
}

void swap(int *px, int *py)
{
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

A correct version using pointers.



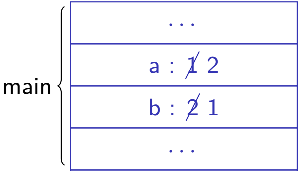
```c
void swap(int *x, int *y);

int main()
{
  int a = 1;
  int b = 2;

  swap(&a, &b);

  return b;
}

void swap(int *px, int *py)
{
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

A correct version using pointers.



```c
void swap(int *x, int *y);

int main()
{
  int a = 1;
  int b = 2;

  swap(&a, &b);

  return b;
}

void swap(int *px, int *py)
{
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

A correct version using pointers.



Pointers

- *A pointer is a variable that contains the address of a variable.*

- declaration: `int *px, *py;` declares two pointer-to-ints.
- address-of operator: `&x` returns a pointer to the variable `x`.
- dereferencing operator: `*px` returns the value that is pointed to.

- `void *px` is a pointer-to-void
  - » Casting to and from other pointer types is permitted.
  - » All pointers have the same size, e.g., `sizeof(void *) = sizeof(int *)`.
  - » A pointer-to-void cannot be dereferenced.

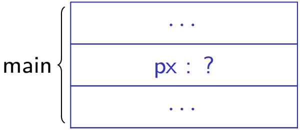
- The `NULL` pointer, `(void *)0`, does not point to anything.
  - » C guarantees that zero is never a valid address for data.
  - » Often used as `None` as in OCaml.

What is wrong with this program?

```
int *f(void);

int main()
{
  int *px;

  px = f();

  return *px;
}

int *f(void)
{
  int x = 7;

  return &x;
}
```

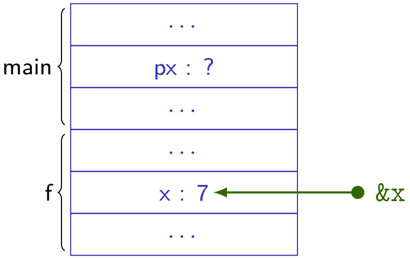
What is wrong with this program?

```c
int *f(void);

int main()
{
  int *px;

  px = f();

  return *px;
}

int *f(void)
{
  int x = 7;

  return &x;
}
```



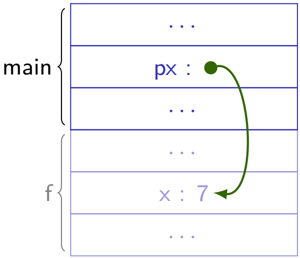
What is wrong with this program?

```c
int *f(void);

int main()
{
  int *px;

  px = f();

  return *px;
}

int *f(void)
{
  int x = 7;

  return &x;
}
```



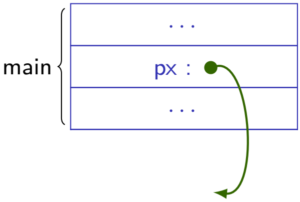
What is wrong with this program?

```
int *f(void);

int main()
{
  int *px;

  px = f();

  return *px;
}

int *f(void)
{
  int x = 7;

  return &x;
}
```



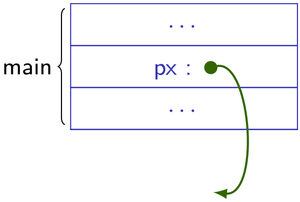
What is wrong with this program?

```c
int *f(void);

int main()
{
  int *px;

  px = f();

  return *px;
}

int *f(void)
{
  int x = 7;

  return &x;
}
```



What is wrong with this program?

```
int *f(void);

int main()
{
  int *px;

  px = f();

  return *px;
}

int *f(void)
{
  int x = 7;

  return &x;
}
```



C11 §6.2.4 (2): "The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime."

Arrays

- An array is a block of consecutive elements.
- declaration: `int a[5];` declares `a` as an array of 5 integers.
- initialize: `char b[] = { 21, 7, 3, 1, 9 };`
  declares `b` as an initialized array of 5 elements.
- subscript: `a[2]` refers to the third element of `a`.

Arrays

- An array is a block of consecutive elements.
- declaration: `int a[5];` declares `a` as an array of 5 integers.
- initialize: `char b[] = { 21, 7, 3, 1, 9 };`
  declares `b` as an initialized array of 5 elements.
- subscript: `a[2]` refers to the third element of `a`.
- An array is represented by a pointer to the first element;
  `int *pa = a` is the same as `int *pa = &a[0]`.
- Pointer arithmetic is defined so as to correspond to array indexing;
  `a[i]` is the same as `*(a + i)`.

```
// copy the contents of b to a
// using array indices
for (i=0; i < 5; ++i)
  a[i] = b[i];
```

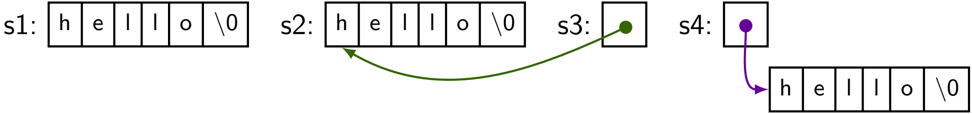
```
// same using pointers
int *pa = a, *pb = b;
for (i=0; i < 5; ++i)
  *(pa++) = *(pb++);
```

In terms of byte addressing: `pa + 1 == (char *)pa + sizeof(int)`.

Strings

In C, a string is a sequence of characters that ends with `'\0'` (null character).

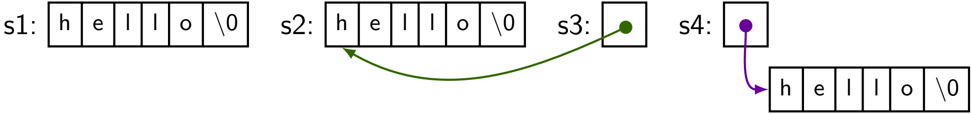
```
char s1[] = { 'h', 'e', 'l', 'l', 'o', '\0' }; // array of 6 chars
char s2[] = "hello"; // array of 6 chars initialized by string literal
char *s3 = s2;
char *s4 = "hello"; // the same?
```



Strings

In C, a string is a sequence of characters that ends with `'\0'` (null character).

```
char s1[] = { 'h', 'e', 'l', 'l', 'o', '\0' }; // array of 6 chars
char s2[] = "hello"; // array of 6 chars initialized by string literal
char *s3 = s2;
char *s4 = "hello"; // the same?
```



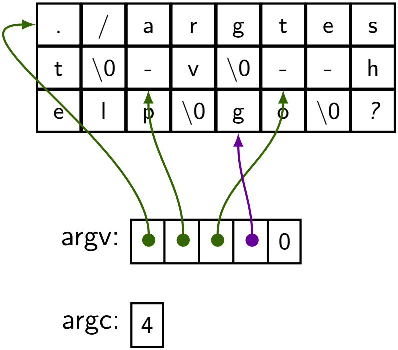

```
void strcpy(char* s, char *t)
{
  while ((*s++ = *t++) != '\0')
    ;
}
```

- Routines in `string.h`: `strncpy` (better), `strcmp`, `strlen`, ...

Command-line arguments: array of strings

```c
#include <stdio.h>

// argc = argument count
// argv = argument vector
// sometimes: char *argv[]
int main(int argc, char** argv)
{
  int i;

  for (i = 0; i < argc; i++)
    printf("argument %d: %s\n",
           i, argv[i]);

  return 0;
}
```

`./argtest -v --help go`



(By convention, `argv` has size `argc + 1` and the last element contains a NULL pointer.)

Environment variables: array of strings

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

extern char **environ;

int main(int argc, char** argv)
{
  char **pv = environ;

  for (; *pv; ++pv) // *pv != NULL
    printf("%s\n", *pv);

  printf("\n%s\n%s\n", getenv("HOME"),
                       getenv("SHELL"));

  return 0;
}
```

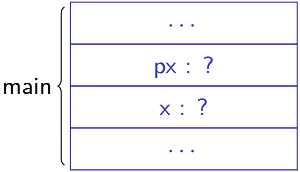
- In a POSIX-compatible environment, `environ` is a (global) variable that points to an array of pointers to strings representing the environment.
- The block of (`VARNAME=VALUE\0`)* strings is constructed by the parent process.
- The last value in the `environ` array is a NULL pointer.
- The `getenv` function takes a name and returns a pointer to the corresponding value, or NULL if the name is not found.

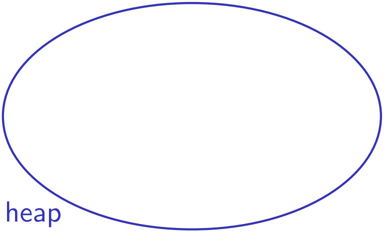
Plan

Fixing the earlier incorrect program.

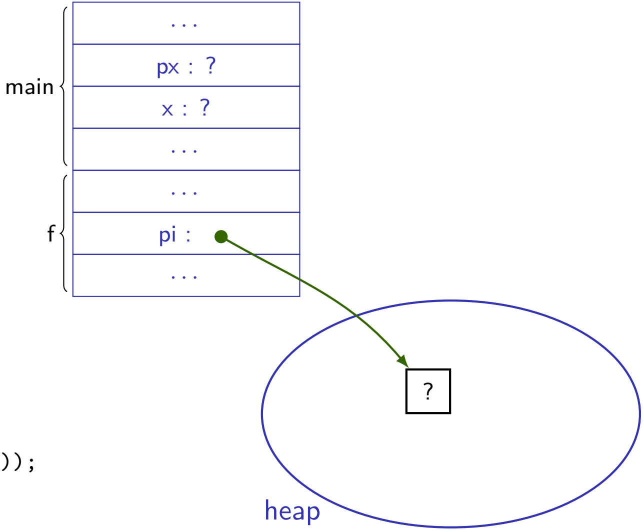
```
int *f(void);

int main()
{
  int *px;

  px = f();

  return *px;
}

int *f(void)
{
  int x = 7;

  return &x;
}
```

```c
#include <stdlib.h>

int *f(void);

int main()
{
  int *px;
  int x;

  px = f();
  x = *px;
  free(px);

  return x;
}

int *f(void)
{
  int *pi = malloc(sizeof(int));
  *pi = 7;

  return pi;
}
```

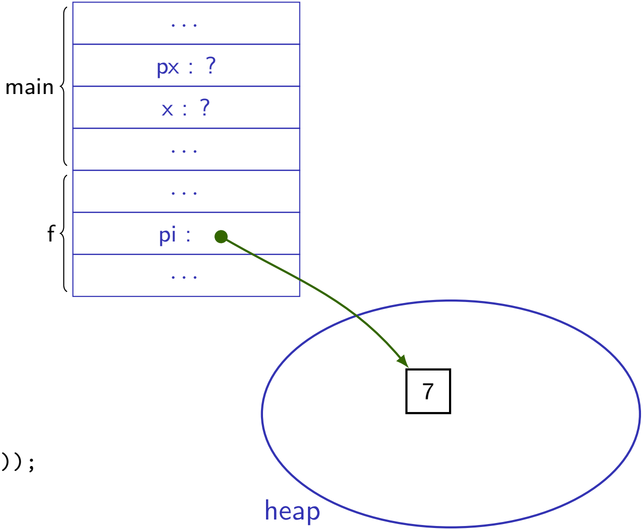




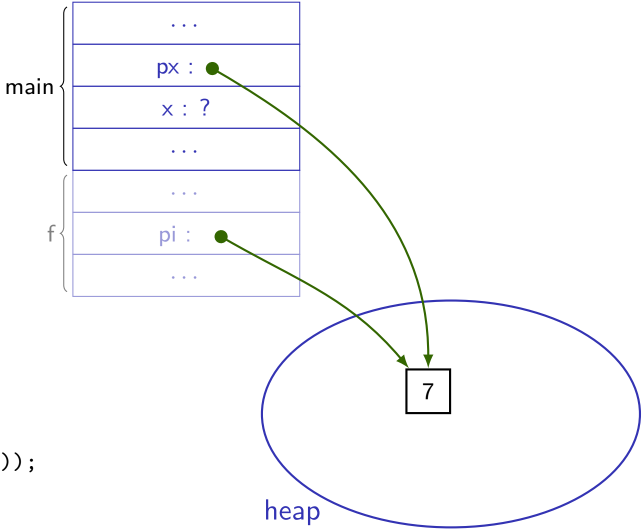
```c
#include <stdlib.h>

int *f(void);

int main()
{
  int *px;
  int x;

  px = f();
  x = *px;
  free(px);

  return x;
}

int *f(void)
{
  int *pi = malloc(sizeof(int));
  *pi = 7;

  return pi;
}
```



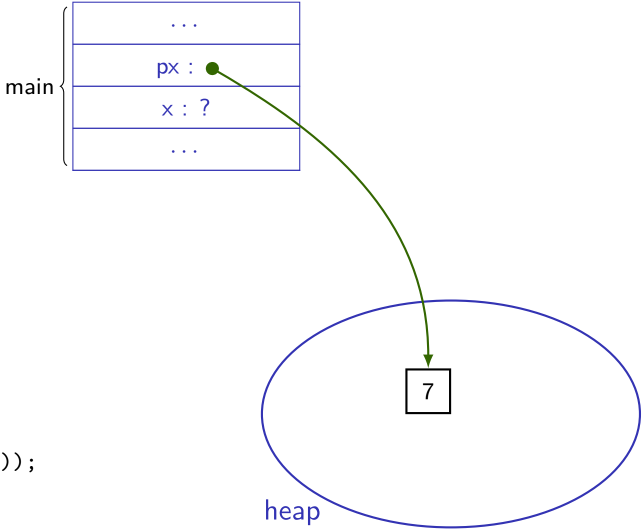
```c
#include <stdlib.h>

int *f(void);

int main()
{
  int *px;
  int x;

  px = f();
  x = *px;
  free(px);

  return x;
}

int *f(void)
{
  int *pi = malloc(sizeof(int));
  *pi = 7;

  return pi;
}
```



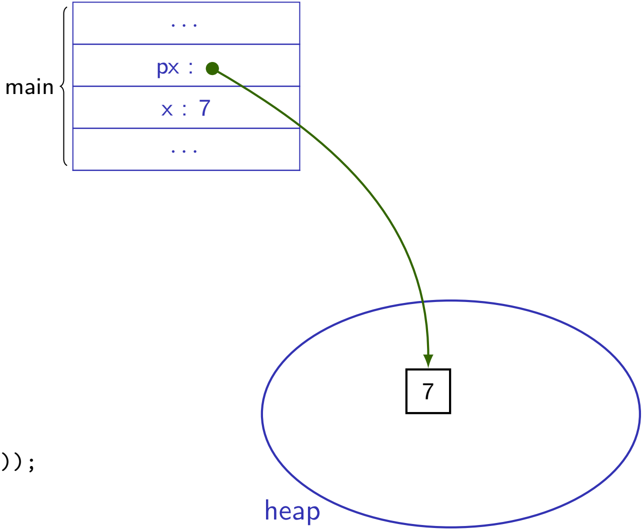
```c
#include <stdlib.h>

int *f(void);

int main()
{
  int *px;
  int x;

  px = f();
  x = *px;
  free(px);

  return x;
}

int *f(void)
{
  int *pi = malloc(sizeof(int));
  *pi = 7;

  return pi;
}
```



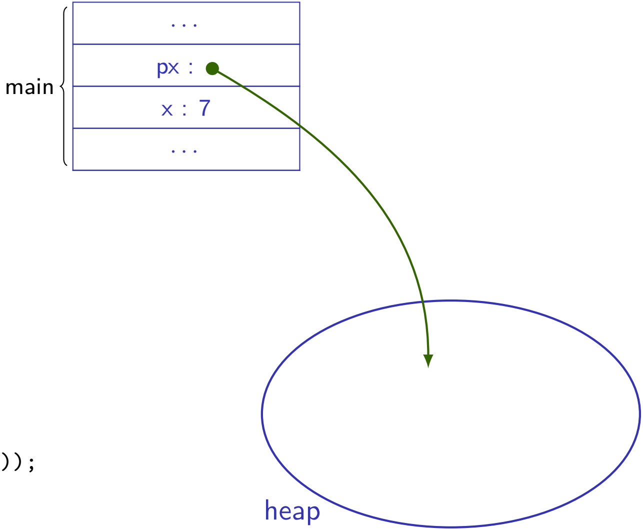
```c
#include <stdlib.h>

int *f(void);

int main()
{
  int *px;
  int x;

  px = f();
  x = *px;
  free(px);

  return x;
}

int *f(void)
{
  int *pi = malloc(sizeof(int));
  *pi = 7;

  return pi;
}
```



```c
#include <stdlib.h>

int *f(void);

int main()
{
  int *px;
  int x;

  px = f();
  x = *px;
  free(px);

  return x;
}

int *f(void)
{
  int *pi = malloc(sizeof(int));
  *pi = 7;

  return pi;
}
```



```c
#include <stdlib.h>

int *f(void);

int main()
{
  int *px;
  int x;

  px = f();
  x = *px;
  free(px);

  return x;
}

int *f(void)
{
  int *pi = malloc(sizeof(int));
  *pi = 7;

  return pi;
}
```



Dynamic memory — allocation on the heap

- `void *malloc(size_t size)`: allocates `size` bytes and returns a pointer to it, or NULL if not possible.
- `void free(void *ptr)`: frees memory earlier allocated by `malloc`.

- Rule 1: Do not access memory after it has been freed.
- Rule 2: One call to free for each call to malloc.

- Dynamic memory is used for creating “objects”:
  - » that outlive the scope in which they are created;
  - » whose size is not known at compile time.

- E.g., create an array of `n` integers:
  ```
  int *px = malloc(n * sizeof(int));
  ```
- Alternative using `void *calloc(size_t nmemb, size_t size)`:
  ```
  int *px = calloc(n, sizeof(int));
  ```
  (`calloc` fills allocated memory with zeroes.)

Plan

Enumerated types

```
enum suit { diamonds, clubs, hearts, spades };

const char* string_of_suit(enum suit s)
{
    switch (s)
    {
        case diamonds: return "diamonds";
        case clubs: return "clubs";
        case hearts: return "hearts";
        case spades: return "spades";
    }
}

int main(void)
{
    enum suit s = diamonds;
    puts(string_of_suit(s));
    return 0;
}
```

- `DIAMONDS` is a `const int`
- By default, first enumerator $= 0$
- And next enumerator is $+1$
- May define value explicitly: `= 7`
- May have duplicate values

Enumerated types

```
enum suit { diamonds = 0, clubs, hearts, spades };

const char* string_of_suit(enum suit s)
{
    static const char* suit_names[] =
      { "diamonds", "clubs", "hearts", "spades" };

    return suit_names[s];
}

int main(void)
{
    enum suit s = diamonds;
    puts(string_of_suit(s));
    return 0;
}
```

Structs

A structure groups together one or more variables of possibly different types.

```
// declare a structure type
struct point {
  int x;
  int y;
};
```

```
// define variables
struct point maxpt = { 640, 320 };
struct point midpt = { .x = 320, .y = 160 };

// refer to elements
int p = maxpt.x * maxpt.y;
midpt.x = 300;
```

Structs

A structure groups together one or more variables of possibly different types.

```
// declare a structure type
struct point {
  int x;
  int y;
};
```

```
// define variables
struct point maxpt = { 640, 320 };
struct point midpt = { .x = 320, .y = 160 };

// refer to elements
int p = maxpt.x * maxpt.y;
midpt.x = 300;
```

Create arrays of structures and pointers to structures.

```
struct point ps[10]; // an array of 10 points

ps[9].x = 10;

// a pointer to an array of n points
struct point *ps = malloc(n * sizeof(structure point));

(*ps).x = 7; // dereference pointer then access field
ps->x = 7; // access field through pointer (same meaning as previous)
```

Typedef

- Introduce type synonyms with `typedef`

```
typedef int Length;

Length len, maxlen;
Length *lengths[];
```

- Frequently used with structures and pointers.

```
typedef char *String;

typedef struct point {
  int x;
  int y;
} Point, *ppoint;

Point maxpt = { 640, 320 };
ppoint pt = malloc(sizeof Point); // sizeof *ppoint
```

- The new type name is in the position of a variable name and *not* directly after the keyword.

Unions

A union specifies overlapping fields. It gives different *views* of an area of memory.

```c
// declare a union type
union bytes {
  unsigned int d;
  char bytes[sizeof(int)];
};
```

```c
int main(void)
{
  union bytes v;

  v.d = 0x0ABADCAFE;

  printf("v.d=0x%x\n", v.d);
  for (int i = 0; i < sizeof(int); ++i) {
    printf("%hhx\n", v.bytes[i]);
  }

  return 0;
}
```

Unions

A union specifies overlapping fields. It gives different *views* of an area of memory.

```c
// declare a union type
union bytes {
  unsigned int d;
  char bytes[sizeof(int)];
};
```

```c
int main(void)
{
  union bytes v;

  v.d = 0x0ABADCAFE;

  printf("v.d=0x%x\n", v.d);
  for (int i = 0; i < sizeof(int); ++i) {
    printf("%hhx\n", v.bytes[i]);
  }

  return 0;
}
```

Used more rarely than `struct`.

Unions: example

```c
#include <stdio.h>

enum variant_label { UINT, FLOAT };

typedef struct {
    enum variant_label label;
    union {
      unsigned int d;
      float f;
    };
} variant;
```

```c
void print(variant v)
{
    switch (v.label)
    {
        case UINT:
            printf("%u\n", v.d); break;

        case FLOAT:
            printf("%f\n", v.f); break;
    }
}

int main(void)
{
    variant v1 = { .label = UINT, .d = 77 };
    variant v2 = { .label = FLOAT, .f = 0.3 };

    print(v1); print(v2);
    return 0;
}
```

```ocaml
type variant =
  | Uint of int
  | Float of float;;
```

Plan

Modules and Abstract Data Types in C

stack.h

```
typedef struct _Stack *Stack;

Stack stack_new(void);
void stack_push(Stack s, void *v);
void *stack_pop(Stack s);
void stack_free(Stack s);
```

- The header file contains the interface: declarations only, no definitions.
- Here there is a *forward declaration* of `struct _Stack`. The declared type is *incomplete*: cannot use in variable declarations or with `sizeof`.
- The pointer type `Stack` can be used in variable declarations and with `sizeof`.
- The `extern` specifier declares variables.

Modules and Abstract Data Types in C

stack.h

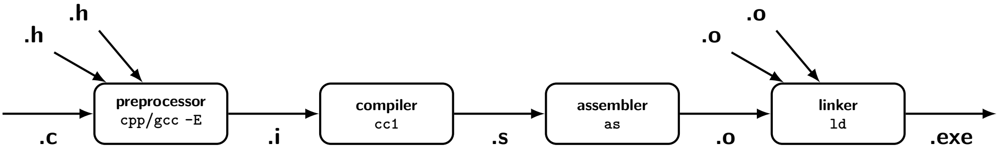
```
typedef struct _Stack *Stack;

Stack stack_new(void);
void stack_push(Stack s, void *v);
void *stack_pop(Stack s);
void stack_free(Stack s);
```

- The source file contains the implementation.
- One definition of each type, variable, and function.
- The `static` specifier hides function and global variable names from the outside.

stack.c

```
#include "stack.h"
#include <stdlib.h>

struct elem {
  void *x;
  struct elem *link;
};

struct _Stack {
  int count;
  struct elem *head;
};

Stack stack_new(void)
{
  Stack s = malloc(sizeof *s);
  s->count = 0; s->head = NULL;
  return s;
}
```

Compiler driver: cc (gcc, clang, ccomp)



- C compilers, like gcc, clang, and CompCert, operate in four main stages.
- `cc -E`: do preprocessing only.
- `cc -S`: stop after compiling to assembler.
- `cc -c`: stop after producing an object file.
- otherwise: compile all files and link symbols to make an executable file.

Preprocessing: source file inclusion (C11 §6.10.2)

- Two forms to include header or source files
  - `#include "file"`
  - `#include <file>`
- Find the file, preprocess it recursively, and replace the directive with the result.

- The file search mechanism is implementation-defined.
- To see where the compiler looks:
  `cpp -v /dev/null` or `clang -x c -v /dev/null`.
- `"includes"` are sought in the same directory as the source file and then as for <include>s.
- Use `-I path` to cons `path` onto the <include> search path (see also `CPATH` variable, `-nostdinc`, `-iquote`, and `-isystem`).

Object files, Libraries, and Symbols

- Object files (`*.o`) contain both compiled functions and a symbol table.
- The `nm` command shows the symbol table. Usual symbol types:
  - » `U`: undefined, e.g., declared and used in a module but not defined.
  - » `T`: function defined in text (code) section.
  - » `B`: variable initialized to zero (bss section).
  - » `D`: variable initialized to another value.
- The `objdump` command shows the contents (use with `-d`/`-s`).

- Object files are grouped together into static (`*.a`) and dynamic libraries (`*.so`).

Linking

The linker resolves symbols by scanning object files and libraries one-by-one in the order given on the command-line. It maintains a list of unresolved references.

- At each object file or library, it tries to find unresolved references.
- Object files specified directly are always imported.
- Object files from within libraries are only imported when required; the linker iterates on a library until no further symbols are resolved.
- After the scan, if any entries are still unresolved, it raises an error.

This explains why libraries are usually specified last.

Object files are specified directly, e.g., `stack.o`.
Library files are specified with `-l`, e.g., `-lm` specifies `libm.a`.
The standard library (`libc.a`) is automatically included.

The `-L` option conses paths onto the list of library search paths.

Plan

Current Working Directory

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

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
  char buf[BUFSIZE];

  getcwd(buf, BUFSIZE);
  printf("current working directory=%s\n", buf);

  chdir("/usr/bin");

  getcwd(buf, BUFSIZE);
  printf("current working directory=%s\n", buf);
}
```

pid, ppid, uid, gid

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  printf("pid=%d\n", (int)getpid());
  printf("ppid=%d\n", (int)getppid());
  printf("uid=%d\n", (int)getuid());
  printf("gid=%d\n", (int)getgid());
}
```

Other features

- The conditional operator (`?:`)
- pointers to functions
- Alignment
- goto and labels
- Non-local jumps (setjmp.h)
- Variable argument lists (stdarg.h)
- Inline functions
- Variable length arrays
- Flexible array members
- The restrict keyword

Operator Precedence [Kernighan and Ritchie (1988): The C Programming Language]

Operators	Associativity
`()` `[]` `->` `.`	left to right
`!` `~` `++` `--` `+` `-` `*` `&` (*type*) `sizeof`	right to left
`*` `/` `%`	left to right
`+` `-`	left to right
`<<` `>>`	left to right
`<` `<=` `>` `>=`	left to right
`==` `!=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=`	right to left
`,`	left to right