Part 2: C/C++ Basics

Contents

[DOCUMENT FINALIZED]

C++ : feature available in C++ but not in C

C vs. C++

```
// this is a C program
#include <stdio.h>


int main()
{
  printf("hello world\n");
  return 0;
}
```

```
// this is a C++ program
#include <iostream>
using namespace std;

int main()
{
  cout << "hello world" << endl;
  return 0;
}
```

Week 2 Similar syntax (how it looks) and semantics (what it does)

C can be considered a C++ subset (well, almost). That is: most C programs are valid C++ programs

Major C++ additions: classes, inheritance, operator overloading, templates

In this course we'll treat C as C++ subset and use g++ as compiler. If you want to compile the presented programs with gcc you need to instruct it to accept c99 standard programs: `gcc -std=c99`

C++ features not available in C will be highlighted.

Basic Building Blocks

```
// this is a comment
/*
  This is also a comment
*/
#include <stdio.h> // preprocessor command

int foo(int x)     // function definition
{                  // code block
  return x+1;      // return expression value
}

int main()         // this is where all C programs start
{
  int i = 0;               // variable definition + init.
  while (i < 10) {         // loop + condition
    i = i+1;               // expression + assignment
    printf("%d ", foo(i)); // function calls, output
  }
  if (i >= 10) i = 1;      // conditional code execution
  else         i = 0;
  return i;                // return result, exit function
}
```


C++ Programming Quickstart

We'd like to start programming right away. Here is what you need to know:

- C++ program execution starts in function `main`
- Statements are executed one after the other
- Variables have to be defined before using them, e.g.
  `int i = 5; // int variable i set to 5`
- Common operators used for arithmetic and Boolean expressions, e.g.
  ```
  a = b + c; // b plus c assigned to a
  a < b      // true if a less than b
  ```
- Code blocks are enclosed in `{...}`
- Conditional program execution:
  ```
  if (a < b) {
     /* executed if condition is true */
  }
  while (a >= b) {
     /* repeatedly executed while true */
  }
  ```


C++ Programming Quickstart (continued)

We also would like to see some program output.

Without going into much detail yet we will use C library function `printf` ("print formatted", type man 3 printf for learn more, including the need to include `stdio.h`).

```
// outputs some text enclosed in ""
printf("hello world");

// output variables
// %d is a placeholder for an int value
// %f is a placeholder for a float value
// number of % and following variables
// must match
int a = 5;
float b = 3.5;
printf("This is an integer: %d", a);
printf("This is a fp value: %f", b);

// move cursor to next line (newline)
printf("\n");
```


C++ Programming Quickstart (continued)

After creating a source code file (e.g. foo.c) with an editor such as emacs, issue

```
g++ foo.c
```

This will create file a.out, which you can start by issuing ./a.out in the terminal window. If you want to give the created executable a different name use option -o

If the program contains errors, the compiler will let you know.

In this case you have to correct the errors using the editor and compile again.

It is a good idea to open two terminal windows: one for editing (you don't leave the editor) and one for compiling and running your program.

In the lab we will see how to compile from within emacs, which is convenient because you can directly jump to error locations.

Identifiers

Used to name variables, functions, struct members, and labels

- Identifiers are case-sensitive
- Start with _ or letter
- Remaining part all letters, digits, or _
- Exceptions are C++ keywords such as
  `if else static for do while ...`

Valid identifiers:

```
sumOfValues x0 FooBar foobar _x_y_z
```

Invalid identifiers:

```
0x $y .name while @abc foo# ^_^ ;-)
```


Comments

It is important to comment your code – for others and yourself!

C++ comments have the following form:

- `// this is a single line comment`
-

```
/* this is a
   multi-
   line
   comment
*/
```

- Multi-line comments cannot be nested!
  Illegal: `/* /* */ */`

Where to put comments?

Good comments are very important. Put them

- at the beginning of files describing their purpose,
- on top of function definitions discussing parameters, function effects, and return values,
- on top of class definitions describing their purpose,
- in front of non-trivial parts, meaning anything you wouldn't instantly understand when looking at the code a month later

No need to write novels or to comment each program statement

Variable Definition

```
int lower, upper, step;
char c;         // all values undefined!
float f = 0;    // initialized with 0
int i = c + 1;  // undefined! Does g++ complain?
const float PI = 3.1415926535;
PI = 0;         // compiler complains! (const)
```

- In C++, variables need to be defined prior to usage
- Variable definitions define the type of data to be stored in a variable. Variables can be initialized on the spot.
- Value of uninitialized variables is undefined! (unlike Java)
- const-qualifier makes variables read-only

Simple Types

Integer Types

- finite range of integral numbers $\{0, \pm 1, \pm 2, ...\}$
- multi-purpose: memory = sequence of integers, everything is encoded as integers

Floating-Point Types

- finite subset of rational numbers of form $m \cdot 2^k$, where m, k are integers
- can express very small to very big numbers
- suitable for scientific computations
- inexact! rounding errors!

```
float a = 1.0;
float b = 1.00000001;
// here, a equals b!
```

- other fundamental algebraic laws no longer valid! E.g. $a + (b + c) \neq (a + b) + c$ for suitable a, b, c

Simple Types (2)

- **bool**: false, true; 1 byte (8 bits) C++
- **char**: ASCII character ('a', '?' ...); 1 byte integer
- **short**: -32,768..+32,767; 2 byte integer (16 bits)
- **int**: -2,147,483,648..+2,147,483,647; 4 byte integer (32 bits)
- **float**: $\approx -10^{38}.. - 10^{-38}, 0, +10^{-38}.. + 10^{38}$ 4 byte floating-point value (7 digits)
- **double**: $\approx -10^{308}.. - 10^{-308}, 0, +10^{-308}.. + 10^{308}$ 8 byte floating-point value (15 digits)

Examples

```
#include <stdio.h>

int main()
{
  bool flag = false;      // 1 byte Boolean variable
  int numOfBeans = 0;     // 4 byte signed int. variable
  unsigned short bits16 = 0; // 2 byte unsigned integer
  float x = 0.1;          // 4 byte f-p variable

  printf("flag=%d numOfBeans=%d bits16=%d PI=%.20f",
         flag, numOfBeans, bits16, x);
}

Output:
flag=0 numOfBeans=0 bits16=0 x=0.10000000149011611938
```

- `printf` prints values of various types defined in the format string by %d (integer) and %.20f (float with 20 digits after decimal point). See `http://www.cplusplus.com/reference/cstdio/printf/` or the section on formatted input/output later in the notes for details.
- Can you explain why the value of x is not 0.1 in the output?

Integer Type Qualifiers: signed, unsigned

Specifies whether a variable is signed or unsigned

No qualifier → signed

• signed char:	-128..127	1 byte
• unsigned char:	0..255	1 byte
• short:	-32768..32767	2 bytes
• unsigned short:	0..65535	2 bytes
• unsigned int:	0..4,294,967,295	4 bytes

Arithmetic and Number Ranges

```
unsigned char foo = 255;
unsigned char bar = foo+1; // bar = 0!
int x = 123456, y = 654321;
int z = x * y;  // z = -824525248  Ouch!
```

No overflows are detected in C++ arithmetic.

Integer +/− simply wraps around!

More specifically: arithmetic is done modulo 2^k (k = variable bit size), i.e. only the remainder when dividing by 2^k is maintained (= the k least-significant bits).

So, double check that arithmetic in your program doesn't exceed variable number ranges!

The C++ compiler and the runtime system can't help you.

When in doubt, add explicit range checks.

Integer Constants

Lecture 4

- An integer constant like 12345 is an int
  `int foo = 12345;`
- Unsigned constants end with u or U
  `unsigned short bar = 60000u;`
- Leading 0 (zero) indicates an octal (base 8) constant (e.g. 037 $= 3 \cdot 8 + 7 = 31$)
  `unsigned short file_permissions = 0666;`
- Leading 0x means hexadecimal
  (base 16, digits: 0..9,a,b,d,e,f)
  E.g. 0x1f = 31, 0x100 = 256, 0xa = 10
  `unsigned int thirty_two_ones = 0xffffffff;`

Floating-Point Constants

- Floating-point constants contain a decimal point (123.4) or an exponent (2e-2 $= 2 \cdot 10^{-2} =$ 0.02) or both
- Their type is double (8 bytes), unless suffixed
- Suffixes f and F indicate float (4 bytes)

```
float two = 2.0; // converted to float
float e = 2.71828182845905f;
```


Character Constants

```
char charx   = 'x';      // = 120
char newline = '\n';     // = 10
char digit1  = '0' + 1;  // = 49 ('1')
char hex     = '\x7f';   // = 127
```

- Characters within single quotes e.g. `'A'` `'%'`
- Characters are stored as 1-byte integers using their ASCII code.
  E.g. `'0'` is represented as 48 (`man ascii`)
- Escape sequences for non-printable characters:
  `'\n'` newline, `'\''` single quote,
  `'\\'` backslash, `'\a'` bell,
  `'\r'` carriage return, `'\xhh'` hexadecimal code

Enumeration Types

```
enum Month { JAN=1, FEB, MAR, APR ...};
// JAN=1 FEB=2 MAR=3 APR=4 ...
enum Month u, v;
u = JAN; v = APR;

enum Answer { NO, YES };
enum Answer a = YES;
int b = a; // legal (1)
a = 1;     // illegal!
a = JAN;   // legal?

// C++ allows:  Month u; Answer a;
// (enum is implicit)
```

- List of named integer constants
- First constant has value 0, next 1, etc.
- Values can be assigned
- Unassigned successors set to previous value + 1
- Names in different enumerations must be distinct. Values need not.

Arithmetic Operators

`+ - * / %` : result type depends on operands

```
int x1 = x0 + delta;
float c = a * b;

int y1 = 8 / 5;  // = 1
int y2 = -8 / 5; // = -1
int y3 = 8 % 5;  // = 3
```

- Division `int / int` rounds towards 0
- `x % y` computes the remainder when `x` is divided by `y` (can not be applied to floating-point values)
- Result of `%` for negative operands is machine dependent, as is the action on overflow

Mixing integers and floating-point values

- Two int operands : integer operation
  - Careful! `(4/5) = 0 !`
  - Division result is rounded towards 0
- One integer and one floating-point operand
  - the integer is silently converted into floating-point format
  - then the floating-point operator is executed
  - `(4.0/5) = (4/0.5) = 0.8`
- Two floats: floating-point operation
  - `(4.0/5.0) = 0.8`

If `x` and `y` are integers and you want to compute the “exact” floating-point ratio you need to “cast types” like so:

```
double ratio = ((double)x)/y;
```

This instructs the compiler to generate code that first converts `x` into a double floating-point number.

Relational Operators

Compare two values with

`>  >=  <  <=  == (equal) != (not equal)`

```
bool v1_eq_v2 = (v1 == v2); // equal?
bool x_ge_0 = (x >= 0); // greater or eq.
bool x = 5;        // != 0 -> true
int a = (1 > 0); // true -> 1
```

Result type bool (values: `true` or `false`)

Watch out: == is equality test, = is assigment!

bool vs. int

- bool is only available in C++
- In integer expressions, bool values are interpreted as 0 (false) or 1 (true)
- int values != 0 are interpreted as true, 0 as false

Useful g++ Flags

`g++ -Wall -Wextra -Wconversion -O test.c`

reports potentially dangerous but valid C++ code such as

```
if (c = 0) ...  // assignment, not equality test
```

or uninitialized variables (for which data-flow analysis is required which is done only when optimizing code: `-O`)

Is the value of `c = 0` in above example true or false?

Logical Operators

```
if (a >= 'a' && a <= 'z').. // a is a lower-case letter
if (a < '0' || a > '9')...  // a is *not* a digit
if (!valid) ...             // true iff valid is false
```

`&&` `||` : Boolean shortcut operators

- evaluated from left to right
- evaluation stops when truth-value is known
- `&&` (shortcut and): evaluation of `(exp1 && exp2)` stops when `exp1` evaluates to false
- `||` (shortcut or): evaluation of `(exp1 || exp2)` stops when `exp1` evaluates to true

`!` : Boolean negation `!`false = true, `!`true = false

(can also be applied to ints: `!`5 = false, `!`0 = true)

Increment & Decrement Operators

```
int a = 0;
a++;   // a now 1
a--;   // a now 0
++a;   // a now 1
--a;   // a now 0

int x = 5;
int y = x++;  // y=5, x=6
int z = ++x;  // z=7, x=7

int n = 3;
x = n + n++;  // undefined!
y = y && n++; // DANGER!
```

- `++` : adds 1 to variable, `--` : subtracts 1
- can be either prefix (`++n`) or postfix (`n++`)
- `++n` increments `n`, value of expression is that of `n` after increment
- `n++` increments `n`, value of expression is original value of `n`

Watch out! If expression terms have side-effects like `++` or function calls, evaluation order may matter!

In this case, split expression like so:

```
int x = 5;
int a = (++x) + (x--);

// case 1. ++x first: a = 6 + 6 = 12
// case 2. x-- first: a = 5 + 5 = 10

// better:
int a = x++;   // evaluate x++ first
               // and then --x
a += --x; // shorthand for a = a + (--x)
```

This works because `;` marks a so-called sequence point at which all previous expressions are fully evaluated before proceeding.

Expressions

Week 3

```
(a+b) * (a-b)          // OK
)a+b(                  // not OK
(a2*x + a1)*x + a0     // OK
a + b + c              // OK, a + b first
a + b * c              // OK, * first
(a >= b) || (c != 1)   // Boolean expr.
```

- Built from variables, constants, operators, and ()
- infix notation (i.e., operator between operands)
- ( ) used for explicit evaluation order, must be balanced
- Operators have fixed arity, associativity & precedence

Assignment Operators

```
int a, b, c;
float d;

a = a + 4;      a += 4;    // equivalent
b = b >> x;     b >>= x;   // equivalent
c = c | 3;      c |= 3;    // equivalent
d = d * (a+1);  d *= a+1;  // equivalent
```

- Set/change value of variable
- Syntax: <variable> = <expression> ;
  (<class> denotes a word in a syntactic class, such as a variable or expression)
- Semantics: expression is evaluated first and its value is assigned to variable
- `v OP= e` is equivalent to `v = v OP (e)`,
  where `OP` is one of `+ - * / % << >> & ^ |`

Operator Precedence

Order of operator evaluation in the absence of ( )

```
b + c * d
// * before +
// same as:  (b + (c * d))

b >= 5 && c <= 6
// >= and <= before &&
// same as: ((b >= 5) && (c <= 6))

a = c+1;
// + before =
// same as a = (c+1);
```

Establishing an operator precedence relation decreases the need for explicit ordering using ( )

If in doubt about operator precedence, use ( ) !

Operator Associativity

The evaluation order of binary operators of the same precedence level

```
a - b - c
// ambiguous! Could mean (a-b)-c or a-(b-c)
// In C++, left to right evaluation: (a-b)-c
// - is called left associative

a = b = c+1;
// assignments are evaluated right to left
// same as: a = (b = (c+1));
// the value of an assignment is the
// value that was assigned
// = is right associative
```

Associativity, Precedence, Arity Table

```
() [] -> .                                  ltr 15 (high)
! ~ ++ -- + - * & (type) sizeof             rtl 14
* / %                                       ltr 13
+ -                                         ltr 12
<< >>                                       ltr 11
< <= > >=                                   ltr 10
== !=                                       ltr 9
&                                           ltr 8
^                                           ltr 7
|                                           ltr 6
&&                                          ltr 5
||                                          ltr 4
?:                                          rtl 3
= += -= *= /= %= &= |= <<= >>=              rtl 2
,                                           ltr 1 (low)
```

- rtl: right (to left) associative, ltr: left (to right) associative
- cyan box: arity 1 (unary operators), all others arity 2 (binary)
- number: precedence level (e.g., == binds tighter than =)

There is no need to memorize this table. For now, just be aware that unary operators bind tighter than binary operators, * / % binds tighter than + - and assignment operators are near the bottom and right associative.

Type Conversions

```
int a; double b;

// Implicit type casting
b = a;    // OK, ints can be represented by doubles
a = b;    // not OK, warning should be issued

// Explicit type casting: convert double into int,
// compiler stays silent
a = (int)b;               // oldest C style
a = int(b);               // older C++ style
a = static_cast<int>(b);  // new C++ style
```

Types of variable and expression must be compatible

The expression value is silently converted to type of variable if possible

Explicit type casts suppress warnings, but precision may be lost!

Floating point numbers are truncated when converted to integers, not rounded!

Program Flow Control

- if-then-else
- switch
- goto
- loops
- functions

if-then-else Statement

```
if (y > x) {
  x = y; // executed if condition is true
}

if (x < 0) {
  sign = -1; // first condition true
} else if (x > 0) {
  sign = +1; // first false, second true
} else {
  sign = 0;  // both false
}
```

If the then/else parts consist of more than one statement, it must be enclosed in { }

Good practice: always use { } irrespective of the number of statements.

This way, when adding statements later, the code will not become incorrect.

Code Indentation

```
// bad indentation
if (x > 0) {
do_this...
   }
else   {
     do_that...
     } y = 0;
```

```
// good indentation
if (x > 0) {
  do_this...
} else {
  do_that...
}
y = 0;
```

Code in if and else branches or code blocks in general must be indented (commonly by 2 or 4 spaces) to improve readability.

Using the tab character is discouraged because when mixing spaces with tabs the text appearance depends on the tab-length, which is usually 8 spaces but can vary.

switch Statement

Multi-way switch dependent on integer value

```
char c; ...
switch (c) {          // integer expression
  case '+':           // integer constant
    result = x + y; // gets here if c == '+'
    break;            // continue at (*) below
  case '-':
    result = x - y; // gets here if c == '-'
    break;            // continue at (*) below
  case 'q','x':
    exit(0);       // gets here if c == 'q' or 'x'
  default:         // all other cases handled here
    cerr << "illegal input" << endl;
    exit(10);
}
// (*)
```

Important: each case should be terminated by a break statement, unless the program leaves the block.

Otherwise, execution will “fall through”,
i.e. the following case's code will be executed next.

goto Statement

```
...
goto jump_location;

...

jump_location:; // resume execution here
```

- Control flow resumes at a specific location marked by a label (identifier)
- Avoid! Goto code is hard to understand and maintain ⇝ “Spaghetti code”

```
// pasta anyone?
int i = 5; goto A;
C:;
printf("confused\n");
B:; i++;
A:; if (i < 10) goto B;
goto C;
```


Loops

- Repeat execution of statements until a condition is met
- Three forms:

  while (<expr>)
     <statement>

  do
     <statement>
  while (<expr>) ;

  for (<init> ; <expr> ; <update>)
     <statement>

  where <statement> is either a single statement, or a block enclosed in { }

while Loop

```
// sum up values 1..100

int s = 0, i = 1;
while (i <= 100) {
  s += i;
  i++;
}
```

- `while` (<expr>) <statement>
- while the expression evaluates to true execute statement repeatedly
- repeated code must be indented

do Loop

```
int s = 0, i = 1;

do {
  s = s+i;
  i++;
} while (i <= 100);
```

- `do` <statement> `while` (<expr>) ;
- first execute statement and loop if expression evaluates to true
- repeated code block must be indented
- unlike while loops, bodies of do loops are executed at least once because the expression is checked at the end.

for Loop

```
int s = 0;

for (int i=1; i <= 100; ++i) {
  s += i;
}
```

`for` (<init> ; <expr> ; <update>)

<statement>

is a shorthand for

```
{
  <init> ;
  while ( <expr> ) {
    <statement>
    <update> ;
  }
}
```


for Loop (continued)

Advantage of for loops: initialization, loop condition, and variable update are co-located which makes the code easier to understand.

Each part can be made more complex by using the comma operator, which simply chains expressions:

```
for (int a=2, b=3;
     a * b < 100;
     a++, b += 2) {
  // do something with a and b
}
```

Here, variables `a,b` are defined, the loop body is executed as long as `a * b < 100` and after each iteration `a` is incremented by 1 and `b` by 2.

for Loop (continued)

```
for (float x = 0; x < 1.0, x += 0.1)
  // do something with x
}
printf("%f\n", x); // compiler complains
                   // x undeclared
```

Variables defined in the initialization part are local to the for loop, i.e. they are not accessible outside.

This is a useful data encapsulation feature, which guards against accidentally reusing data that was set elsewhere.

```
double x;
for (x = 0; x < 1.0, x += 0.1)
  // do something with x
}
printf("x after termination %f\n", x);
```

Sometimes, however, we'd like to have access to loop variables after the loop has terminated. In this case, we can just use a previously defined variable.

Loop Control

- `break;` : exits loop immediately
- `continue;` : skips loop body

```
while (...) {
  ...
  break;
  // equivalent to
  // goto break_loc;
  ...
}
break_loc: ;
```

```
while (...) {
  ...
  continue;
  // equivalent to
  // goto cont_loc;
  ...
  cont_loc: ;
}
```

In for loops, continue resumes with the update

Example

```
int N = 1000; // number of items
int x = 5;    // element to look for
int i;

for (i=0; i < N; i++) {
  // look for x at position i
  ...
  if (found)
    break; // loop terminates if found
}

if (i < N) {
  // found x at position i
} else {
  // here: i = N, i.e. we didn't find x
}
```


Functions

```
void sub_task1() { ... } // implemented by Peter

void sub_task1() { ... } // implemented by Ann

void sub_task3() { }     // mock implementation

bool done() { ... }      // implemented by Sue

int main()
{
  sub_task1();  // executes code in function
                // when done, resume here
  sub_task2();

  while (!done()) {
    sub_task3();
  }
}
```

Functions (2)

- Modular programming: break tasks down into smaller sub-tasks
- Idea: give code block a name, jump to it when needed, and resume execution at calling site when done.
- Increases readability
- Eases debugging and program maintenance because program pieces can be tested individually
- Faster project development: each team member can work on a different function.

Function Declarations

```
void write(float x); // returns nothing
int add4(int a1, int a2, int a3, int a4);
int random(); // returns a random number
```

- Functions must be declared before they are used
- Syntax:

  <type> <function-name> (<parameter-list>);
- Meaning: a function is declared that computes and returns a value of a certain type given a list of parameters
- return type void indicates that nothing is returned
- empty parameter list: no parameters are used

Once the compiler sees a function declaration it can check whether subsequent calls meet its standards, i.e., whether the passed on parameters have the correct type and whether the returned value is used according to its type.

E.g., the header file `stdio.h` contains the declaration of function `printf`.

Function Definitions

```
// compute sum of four parameters
int add4(int a1, int a2, int a3, int a4)
{
  return a1 + a2 +a3 + a4;
}
```

- Function definitions specify the code that is executed when calling the function.
- Syntax:

  <type> <name> (<param-list>)
  {
     <statements>
  }
- Exit void functions with `return;`.
  Can be placed anywhere in the function body.
- Values are returned by `return` <expr> `;`
  Type of expression must match function return type.
- Parameters are treated as local variables

Function Examples

```
// compute the square of x
int square(int x) { return x * x; }

// return true if and only if x is odd
bool is_odd(int x) { return x % 2; }

// compute absolute value of x
int abs(int x)
{
  if (x >= 0)
    return x;

  return -x;
}

// does abs(x) return the correct value
// for all x?
```

More Function Examples

```
// compute n! = 1*2*3*4*...*(n-1)*n
// ("n factorial") iteratively
int factorial(int n)
{
  int prod = 1;
  for (int i=2; i <= n; ++i) {
    prod *= i;
  }
  return prod;
}

// compute n! recursively using
// 0! = 1  and  n! = n * (n-1)!
int rfactorial(int n)
{
  if (n <= 1)
    return 1;

  return n * rfactorial(n-1);
}
```


Some C Library Functions

```
// exit program with error code err
// (0 usually indicates success)
// can be queried using shell variable $?
// e.g.:  ls asdfasdf; echo $?
void exit(int err);

// read a character from standard input
int getchar();

// compute the sine of x
double sin(double x);

// return nearest integer to x
double round(double x);
```

To learn more about standard C library functions consult `http://www.cplusplus.com/ref/` or

`man stdio.h / stdlib.h / string.h / math.h`

More about library functions later in the course.

Variable Scope

- Variables (and constants) have a lifespan from the time they are created until they are no longer used
- Local variables are declared within statement blocks enclosed in { }
- They live from the time the block is entered until the block is left, i.e. they are unknown outside the block.
- Memory for them is allocated on the process stack.
- Unlike Java and Python, numerical variables (char, ..., double) are not automatically initialized in C++! It is the programmer's responsibility to ensure that variables that are used are properly initialized.
- When functions are exited, memory for local variables is released

Local Variable Scope

```
int main()
{
  int   uninitialized;
  float initialized = 22.0/7.0; // (*)
  float x = 2.0; // (**)

  { // nested block
    float x; // (***)

    x = initialized; // copies (*) to (***)
  }

  x = 3.1415926; // changes (**)

  for (int i=10; i >= 0; --i) { printf("?"); }
  i = 5; // i unknown here! i local to for block

  int i; // variables can be defined anywhere
  for (i=10; i >= 0; --i) { }
  printf("%d\n", i); // i lives here! value is -1
}
```

Function Call Mechanism

- Uses process stack (Last-In-First-Out data structure)
- Stack-pointer register (SP) in CPU points to next available byte in memory

Calling a function step by step:

- Evaluate parameters and push values onto stack
- Then push return address onto stack
- Then make room for local variables by adjusting SP
- Before returning from the function, store result in register for the caller to be used
- Finally, pop local variables, parameters, and return address off stack, and jump to the return address to continue execution.

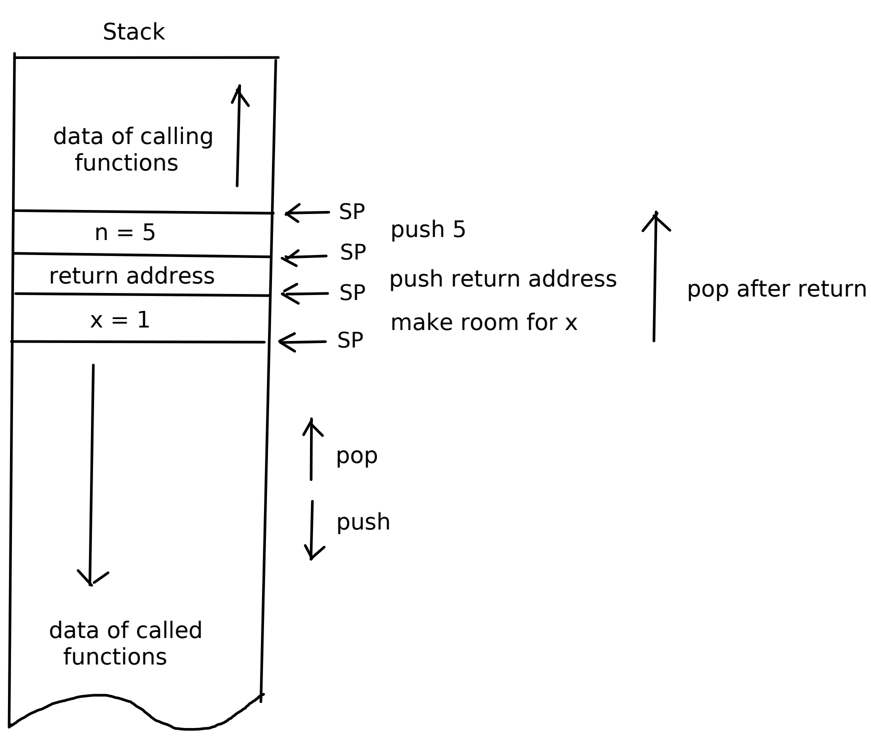
(This is a simplified description of what actually is going on in modern CPUs when calling a function)

Example

```
int foo(int n)
{
  int x = 1;
  return x + n;
}

int a = foo(5);
```

1. parameter 5 is pushed onto stack (parameter n)
2. return address is pushed onto stack (when returning the execution will resume with storing the function result into variable a)
3. room is created for local variable x on stack
4. x is set to 1
5. x + n is evaluated and result 6 is stored in register
6. CPU pops x, return address, and n off the stack, and
7. resumes execution with storing return-value register in variable a (6)



For each function invocation memory for parameters, the return address, and local variables is allocated on the stack.

Stack-based memory allocation is fast — only the SP register has to be changed.

Passing Parameters: Call-By-Value

```
void increment(int x) { ++x; }

int y = 5;
increment(y);

// oops, that didn't work: y is still 5!
```

When a function `increment` is called via `increment(e)` expression `e` is evaluated and its value is copied into local variable `x`

Statements in the function body act on this local copy and do not change values in the evaluated expression `e`

The function is said to have no side effects on the caller's environment.

So it can't possibly change y in this example.

Passing Parameters: Call-By-Reference C++

```
void increment(int &x) { ++x; }

int y = 5;
increment(y);
// that worked: y now 6
```

- A reference to a variable is passed to a function (in form of a memory address)
- Statements in the function body that act on the parameter variable (x) change the variable that has been passed to the function (y).
- This means that functions now can have side effects, i.e. can change something in the caller's environment.
- Can only pass variables, but not expressions, because an address is required. E.g., this call is illegal:

```
increment(3 + x);
```


Swap Function C++

```
void naive_swap(int &x, int &y)
{
  x = y;
  y = x;
}

int a = 1, b = 2;
naive_swap(a, b);  // oops: a = b = 2 !

void swap(int &x, int &y)
{
  // triangle exchange
  int temp = x; x = y; y = temp;
}

a = 1; b = 2;
swap(a, b);         // ok! a = 2, b = 1
```


Passing Large Objects C++

```
void do_something(T big) { ... }
...
T x;
do_something(x); // slow!
```

- Passing large objects of type T by value is wasteful: they are copied into local variables
- Better: const reference

```
void do_something(const T &big) { ... }
...
T x;
do_something(x); // much faster!
```

This is much faster because only an address is passed to the function instead of a big object.

const ensures that we don't accidentally change the variable with which the function was called.

Pros & Cons

Call-by-Value

+ Callee detached from caller, no direct side-effects because parameter values are copied into local variables

− Data is copied to a local variable.
  Can be time consuming

Call-by-Reference

− Side Effects; need to look at function declaration to see whether call-by-reference is used and the function possibly changes parameter variables

− Parameters restricted to variables

+ Only reference (i.e., address in memory) is copied. Fast.
  (const qualifier protects read-only parameters)