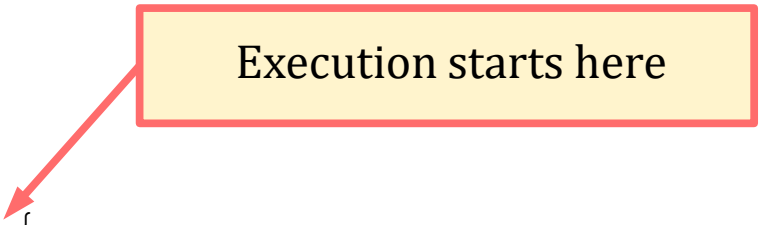
C Programming
Introduction

What is C?

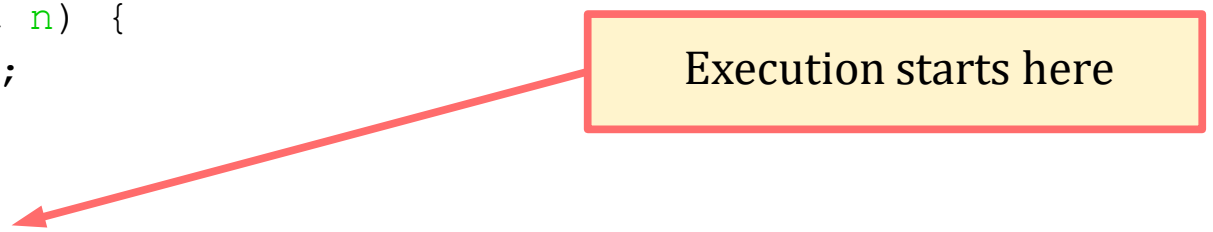
- Imperative programming
- language Statically typed
- Low-level
- Ubiquitous
- Created by Dennis Ritchie in early 1970s to be the language for UNIX
- Standardized in C89 (ANSI C), C99 and C11
- Inspiration for many other languages, e.g., C++, Objective-C, Java, C#
- Good FAQ at http://c-faq.com/

```c
#include <stdio.h>

int x = 5; /* x is a global variable */

/*
 This is a function called square. It takes a single int parameter and
 returns an int.
 */
int square(int n) {
    return n*n;
}

int main(int argc, char **argv) {
    int n = 16;

    printf("The square of %d is %d.\n", x, square(x));
    printf("The square of %d is %d.\n", n, square(n));

    if (x > n) {
        printf("%d is larger than %d.\n", x, n);
    } else {
        printf("%d is not larger than %d.\n", x, n);
    }

    return 0;
}
```

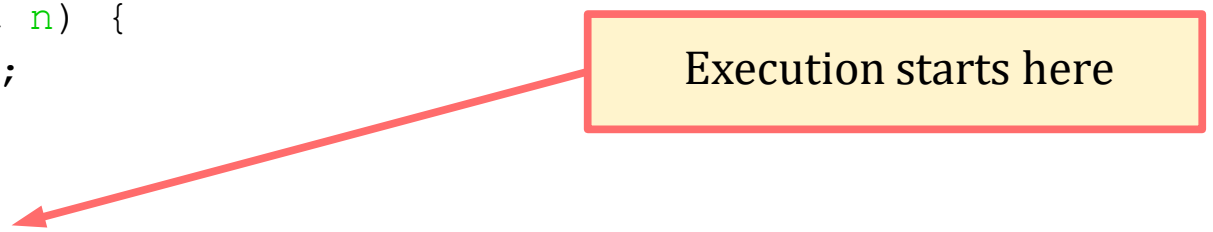
```c
#include <stdio.h>

int x = 5; /* x is a global variable */

/*
 This is a function called square. It takes a single int parameter and
 returns an int.
 */
int square(int n) {
    return n*n;
}

int main(int argc, char **argv) {
    int n = 16;

    printf("The square of %d is %d.\n", x, square(x));
    printf("The square of %d is %d.\n", n, square(n));

    if (x > n) {
        printf("%d is larger than %d.\n", x, n);
    } else {
        printf("%d is not larger than %d.\n", x, n);
    }

    return 0;
}
```



```c
#include <stdio.h>

int x = 5; /* x is a global variable */

/*
 This is a function called square. It takes a single int parameter and
 returns an int.
 */
int square(int n) {
    return n*n;
}

int main() {
    int n = 16;

    printf("The square of %d is %d.\n", x, square(x));
    printf("The square of %d is %d.\n", n, square(n));

    if (x > n) {
        printf("%d is larger than %d.\n", x, n);
    } else {
        printf("%d is not larger than %d.\n", x, n);
    }

    return 0;
}
```



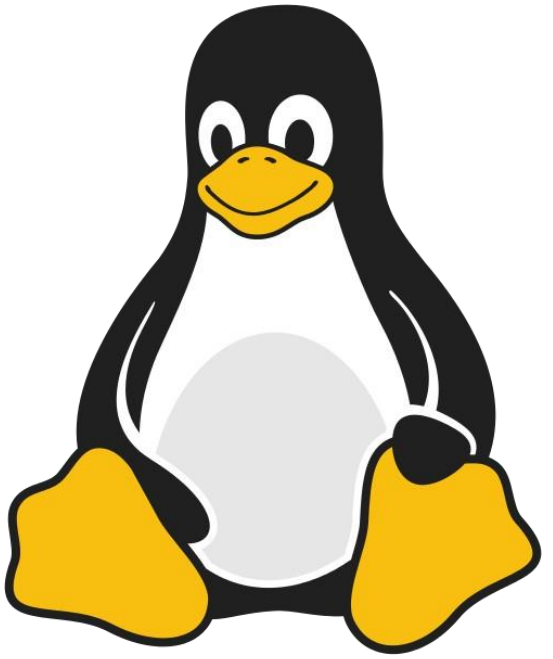
```c
#include <stdio.h>

int x = 5; /* x is a global variable */

/*
 This is a function called square. It takes a single int parameter and
 returns an int.
 */
int square(int n) {
    return n*n;
}

int main() {
    int n = 16;

    printf("The square of %d is %d.\n", x, square(x));
    printf("The square of %d is %d.\n", n, square(n));

    if (x > n) {
        printf("%d is larger than %d.\n", x, n);
    } else {
        printf("%d is not larger than %d.\n", x, n);
    }

    return 0;
}
```




```
The square of 5 is 25.
The square of 16 is 256.
5 is not larger than 16.
```

Getting used to Linux

- Head to a computer lab room with Linux computers (top floor in house 1 or 2) and play around a bit.
- Follow some tutorial online. For example https://ryanstutorials.net/linuxtutorial/
- Try SSH'ing to one of the department's Linux servers from a laptop etc.



Defining variables

- `type_name var_name;`
- `type_name var_name = initial_value;`

- `int x;`
- `char c = 'A';`
- `unsigned long long bignum =`
  `10000000000000;`

Defining variables

- `type_name var_name;`
- `type_name var_name = initial_value;`

- `int x;`
- `char c = 'A';`
- `unsigned long long bignum =`
  `100000000000;`

Note 1: A variable is defined only in its scope

Defining variables

- `type_name var_name;`
- `type_name var_name = initial_value;`

- `int x;`
- `char c = 'A';`
- `unsigned long long bignum = 100000000000;`

Note 1: A variable is defined only in its scope

Note 2: Reading an uninitialized variable (unless global or `static`) leads to ***undefined*** behavior

Undefined behavior

- Many “bad” things in C (reading an uninitialized variable, division by zero etc.) lead to *undefined behavior.*
- In these cases, **anything can happen**!

Undefined behavior

- Many “bad” things in C (reading an uninitialized variable, division by zero etc.) lead to *undefined behavior.*
- In these cases, **anything can happen**!

Undefined behavior

- Many “bad” things in C (reading an uninitialized variable, division by zero etc.) lead to *undefined behavior.*
- In these cases, **anything can happen**!

C is not safe!

Undefined behavior

C FAQ: Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.

Undefined behavior

C FAQ: Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), **or it may fortuitously do exactly what the programmer intended.**

Undefined behavior

C FAQ: Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), **or it may fortuitously do exactly what the programmer intended.**

My view: If the program is incorrect, you want it to crash already during testing!

Built-in integer types

`Type name`	**Size**	**Notes**
`char`	At least 8 bits	The smallest addressable unit that can contain a single character
`short`	At least 16 bits	
`int`	At least 16 bits	The “default” integer type
`long`	At least 32 bits	
`long long`	At least 64 bits	Only since C99

Built-in integer types

`Type name`	**Size**	**Notes**
`char`	At least 8 bits	The smallest addressable unit that can contain a single character
`short`	At least 16 bits	
`int`	At least 16 bits	The "default" integer type
`long`	At least 32 bits	
`long long`	At least 64 bits	Only since C99

Note: Each can be specified as `signed` or `unsigned`

Built-in integer types

Type name	Size	Notes
`char` (*neither* by default)	At least 8 bits	The smallest addressable unit that can contain a single character
`short` (`signed` by default)	At least 16 bits	
`int` (`signed` by default)	At least 16 bits	The "default" integer type
`long` (`signed` by default)	At least 32 bits	
`long long` (`signed` by default)	At least 64 bits	Only since C99

Note: Each can be specified as `signed` or `unsigned`

Other integer types

- Since C99, there are more types defined in `stdint.h`
- Usually better to include `inttypes.h` for some extras

Type name	**Size**	**Notes**
`intN_t` `uintN_t`	Exactly `N` bits (`N` = 8, 16, 32, 64, ?)	Only available if possible for the implementation
`int_leastN_t` `uint_leastN_t`	At least `N` bits `N` = 8, 16, 32, 64, ?	The *smallest* integer type available with at least `N` bits
`int_fastN_t` `uint_fastN_t`	At least `N` bits `N` = 8, 16, 32, 64, ?	The *fastest* integer type available with at least `N` bits

Built-in floating-point types

`Type name`	**Size**	**Notes**
`float`	Usually 32 bits	Usually IEEE-754 single precision floating point
`double`	Usually 64 bits	Usually IEEE-754 double precision floating point
`long double`	At least the size of `double`	?

`sizeof`

- To find out the size in memory of any data type, you can use the `sizeof` operator
- `sizeof` gives the size in units of the size of `char`

  → `sizeof(char)` is 1 by definition
- The given value is of the unsigned integer type `size_t`

Arrays

- An array is a **fixed-size** sequence of elements of the same type
- Array elements are always stored contiguously in memory
- There is no string type, C uses arrays of `char`

Arrays

- An array is a **fixed-size** sequence of elements of the same type
- Array elements are always stored contiguously in memory
- There is no string type, C uses arrays of `char`

Example array definitions:

```
int arr1[10];
int arr2[10] = {1,2,3,4,5};
int arr3[] = {1,2,3,4,5};
char s[] = "Hello";
```

Arrays

- An array is a **fixed-size** sequence of elements of the same type
- Array elements are always stored contiguously in memory
- There is no string type, C uses arrays of `char`

Example array definitions:

```
int arr1[10];
int arr2[10] = {1,2,3,4,5};
int arr3[] = {1,2,3,4,5};
char s[] = "Hello";
```

This syntax is only possible during initialization!

Arrays,cont.

- Every element in an array can be read and written to independently
- Trying to read or write outside the bounds of an array *hopefully* crashes the program – **C performs no bounds checking!**

Arrays,cont.

- Every element in an array can be read and written to independently
- Trying to read or write outside the bounds of an array *hopefully* crashes the program – **C performs no bounds checking!**

Example array uses:

```
int arr[] = {10,11,12,13,14};
int x =arr[0]; /*  Setsx to  10 */
arr[4]=42; /*   Writes 42to  arr[4] */
arr[5] = 42; /* Anythinghappens!  */
```

Arrays,cont.

- Every element in an array can be read and written to independently
- Trying to read or write outside the bounds of an array *hopefully* crashes the program – **C performs no bounds checking!**

Example array uses:

```
int arr[] = {10,11,12,13,14};
int x =arr[0]; /*  Setsx to  10 */
arr[4]=42; /*   Writes 42to  arr[4] */
arr[5] = 42; /* Anythinghappens!  */
```

Multidimensional arrays

- You can create *n*-dimensional arrays for any $n \geq 1$
- In memory these look just like a single large array, but the compiler will calculate the indices for you

Example:

```
int arr1[3][2] = {{0,1},{2,3},{4,5}};
/* arr2 is exactly like arr1 in memory */
int arr2[6] = {0,1,2,3,4,5};
arr1[2][0]; /* Evaluates to 4 */
```

Pointers (very briefly)

- A pointer is a memory address
- Pointers can be stored in variables of pointer type
- For any type `t`, there is a corresponding pointer type `t*`

```
int a = 5; /* a is an integer */
int* ptr; /* ptr is a pointer to an integer */
```

A warning on syntax

```
int* a, b;
```

is the same as

```
int* a;
int b;
```

A warning on syntax

```
int* a, b;
```

is the same as

```
int* a;
int b;
```

Writing the `*` next to the variable name instead is preferred for a bit of added clarity.

```
int *a, b; /* only a is a pointer */

int *a, *b; /* a and b are pointers */
```

Pointers (very briefly)

- Using *referencing* (the `&` operator) we can get a pointer to any variable
- Using *dereferencing* (the `*` operator) we can get the value stored at the address pointed to by a pointer

```
int a = 5; /* a is an integer */
int *ptr; /* ptr is a pointer to an integer */
ptr = &a; /* The value of ptr is a's address */
*ptr = *ptr + 2; /* a is now 7 */
```

Arithmetic expressions

- Basic expressions: `+`, `-`, `*`, `/`, `%`

  → work as expected

Short forms:

- `n++` sets `n=n+1` and evaluates to *old* value of `n`
- `++n` sets `n=n+1` and evaluates to *new* value of `n`
- `n--` and `--n` similar
- `n*=3` is equal to `n=n*3` etc.

Boolean expressions

- There is no boolean type in C (well, there is in C99)
- Boolean expressions evaluate to an `int`

  → `0` is interpreted as *false*

  → everything else is interpreted as *true*
- E.g., `if(42)` and `if(-3)` will take the `if`-branch,
- `if(0)` will not

Boolean expressions

- Comparisons: ==, !=, <, >, =<, =>

  → work as expected
- Conjunction: &&
- Disjunction: ||
- Negation: !

Boolean expressions

- Comparisons: ==, !=, <, >, =<, =>

  → work as expected
- Conjunction: &&
- Disjunction: ||
- Negation: !

Warning: Don't mix up = with ==, `&` with `&&`, or `|` with `||`

A C compiler will happily let you write things like:

```
if (a = 5) {
   // do something if a equals 5
}
```

if-else statements

```
if (a == 5) {
   // do something when a == 5
} else if (a > 0) {
   // do something when a > 0, but a != 0
} else {
   // do something when a <= 0
}
```

while loops

```
int n = 4096;

while (n >= 1) {
  // do something
  n = n / 2;
}
```

for loops

```
int i;

for (i = 0; i < 10; i++) {
   // do something
}
```

for loops

```
int i;

for (i = 0; i < 10; i++) {
   // do something
}
```

for loops

```
int i;

for (i = 0; i < 10; i++) {
   // do something
}
```

```
int i;

i = 0;
while (i < 10) {
   // do something
   i++;
}
```

`break` and `continue`

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

`break` and `continue`

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

```
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
  printf("%c\n", s[i]);
}
```

`break` and `continue`

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

```
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
   printf("%c\n", s[i]);
}
```

```
H
e
l
l
o

W
o
r
l
d
!
```

`break` and `continue`

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

```
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
   if (s[i] == 'l') {
      break;
   }
   printf("%c\n", s[i]);
}
```

`break` and `continue`

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

```
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
   if (s[i] == 'l') {
      break;
   }
   printf("%c\n", s[i]);
}
```

```
H
e
```

`break` and `continue`

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

```
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
   if (s[i] == 'l') {
      continue;
   }
   printf("%c\n", s[i]);
}
```

`break` and `continue`

- The `break` statement immediately exits the innermost loop
- The `continue` statement immediately exits the current iteration of the innermost loop

```
char s[] = "Hello World!";
int i;
for (i = 0; s[i] != 0; i++) {
   if (s[i] == 'l') {
      continue;
   }
   printf("%c\n", s[i]);
}
```

```
H
e
o

W
o
r
d
!
```

More...

- `do-while` loops
- `switch`
- statements `goto`s

Functions

- Functions are declared with a return type and types for all parameters

  → use `void` as return type if the function returns nothing

Functions

- Functions are declared with a return type and types for all parameters

  → use `void` as return type if the function returns nothing

```
void print_n_times(char c, int n) {
   int i;
   for (i = 0; i < n; i++) {
      printf("%c\n", c);
   }
}
```

Functions

- Functions are declared with a return type and types for all parameters

  → use `void` as return type if the function returns nothing

```
void print_n_times(char c, int n) {
   int i;
   for (i = 0; i < n; i++) {
      printf("%c\n", c);
   }
}

int is_ascii_lowercase(char c) {
   if (c >= 'a' && c <= 'z') {
      return 1;
   }
   return 0;
}
```

Functions, cont.

- Arguments to functions are pass-by-value

  → use pointers when you want pass-by-reference

Functions, cont.

- Arguments to functions are pass-by-value

  → use pointers when you want pass-by-reference

```
#include <stdio.h>

void add_two(int n) {
   n += 2;
}

int main() {
   int a = 10;
   add_two(a);
   printf("%d\n", a); /* Prints 10 */

   return 0;
}
```

Functions, cont.

- Arguments to functions are pass-by-value

  → use pointers when you want pass-by-reference

```
#include <stdio.h>

void add_two(int *n) {
   *n += 2;
}

int main() {
   int a = 10;
   add_two(&a);
   printf("%d\n", a); /* Prints 12 */

   return 0;
}
```

`printf` and `scanf`

- Provides *formatted* input and output, respectively.
- Declared in `stdio.h` (use `#include <stdio.h>`)
- The first argument should be a *format string* which specifies how the remaining arguments are to be printed

`printf` and `scanf`

- Provides *formatted* input and output, respectively.
- Declared in `stdio.h` (use `#include <stdio.h>`)
- The first argument should be a *format string* which specifies how the remaining arguments are to be printed

```
int num = 37;
char ch = 'P';
double pi = 3.14159;

printf("The value of num is %d\n", num);
printf("%c follows %c in the alphabet\n", ch+1, ch);
printf("pi is approximately %f\n", pi);
```

`printf` and `scanf`

- Provides *formatted* input and output, respectively.
- Declared in `stdio.h` (use `#include <stdio.h>`)
- The first argument should be a *format string* which specifies how the remaining arguments are to be printed

```
int num = 37;
char ch = 'P';
double pi = 3.14159;

printf("The value of num is %d\n", num);
printf("%c follows %c in the alphabet\n", ch+1, ch);
printf("pi is approximately %f\n", pi);
```

`printf` and `scanf`

- Provides *formatted* input and output, respectively.
- Declared in `stdio.h` (use `#include <stdio.h>`)
- The first argument should be a *format string* which specifies how the remaining arguments are to be printed

```
int num = 37;
char ch = 'P';
double pi = 3.14159;

printf("The value of num is %d\n", num);
printf("%c follows %c in the alphabet\n", ch+1, ch);
printf("pi is approximately %f\n", pi);
```

```
The value of num is 37
Q follows P in the alphabet
pi is approximately 3.141590
```

Some `printf` format identifiers

Identifier	Notes	Example
`%d`	Prints an `int` in decimal	`printf("%d is a number", 5);` → `5 is a number`
`%o` / `%x` / `%X`	Prints an `int` in octal / hexadecimal / HEXADECIMAL	`printf("%o %x %X", 59, 59, 59);` → `73 3b 3B`
`%.`*n*`f`	Prints a `float` or `double` with *n* digits after the point	`printf("%.3f", 10.0 / 3);` → `3.333`
`%c`	Prints a `char` as a character	`printf("%c %c", 'A', 66);` → `A B`
`%s`	Prints a string	`printf("I have %s cats", "two");` → `I have two cats`

There are many more options, check documentation or the web!

Some `scanf` format identifiers

Identifier	Notes	Example
`%d`	Reads an `int` in decimal	`int a;` `scanf("%d", &a);`
`%f` / `%lf`	Reads a `float` / `double`	`double a;` `printf("%lf", &a);`
`%c`	Reads a `char` as a character	`char a;` `scanf("%c", &a);`
`%s`	Reads a string, will automatically append a null char	`char a[10];` `scanf("%s", a);`

Some `scanf` format identifiers

Identifier	Notes	Example
`%d`	Reads an `int` in decimal	`int a;` `scanf("%d", &a);`
`%f` / `%lf`	Reads a `float` / `double`	`double a;` `printf("%lf", &a);`
`%c`	Reads a `char` as a character	`char a;` `scanf("%c", &a);`
`%s`	Reads a string, will automatically append a null char	`char a[10];` `scanf("%s", a);`

Warning: `printf` and `scanf` (and many more C functions) are unsecure when used naively. A bit more on this next time.

Using `ssh` to our servers

- You can use Secure Shell (`ssh`) to connect to the department's UNIX servers from your own computer.
- Some instructions on https://www.it.uu.se/datordrift/faq/ssh
- List of available Linux hosts on https://www.it.uu.se/datordrift/maskinpark/linux
- List of available Solaris hosts on https://www.it.uu.se/datordrift/faq/unixinloggning

Compiling with `gcc`

- The default C compiler on Linux is usually the GNU Compiler Collection (GCC)
- → Also compiles other languages: C++, Objective-C, Ada...
- To invoke GCC from a terminal and compile `myfile.c`:
- ```
  gcc myfile.c -o myfile
  ```
- To run your newly compiled program:
- ```
  ./myfile
  ```
- A good idea to include optional *flags* to tell `gcc` how to behave, for example:

`-Wall`	enables "all" warning messages
`-std=c11`	enables the extensions in the C11 standard

```
gcc -Wall -std=c11 myfile.c -o myfile
```