A History Lesson

Development of language by Dennis Ritchie at Bell Labs culminated in the C language in 1972.

Motivation was to facilitate development of systems software, especially OS development.

Traditionally, supports a procedural view of problem analysis.

Formal language Standard adopted in 1990; required compromises because of vast body of existing C code based on a more-or-less common understanding of the language.

Significant revision, ISO/IEC 9899:1999 or simply C99 if you like, was adopted in 1999.

My presentation will be based on the C99 Standard… most C compilers now support most of that Standard.

The First Program

Since tradition demands it:

```
#include <stdio.h>          // load declarations of std
                            // library functions for I/O

int main() {                            // mandatory fn

   printf("Hello, world!\n");           // output to console

   return 0;                            // exit fn (& pgm)
}
```

Note: `#include` loads declarations from standard C library (and more)

Every C program must have a non-member fn called `main()`.

`main()` must be declared with a return type of `int`.

The Preprocessor

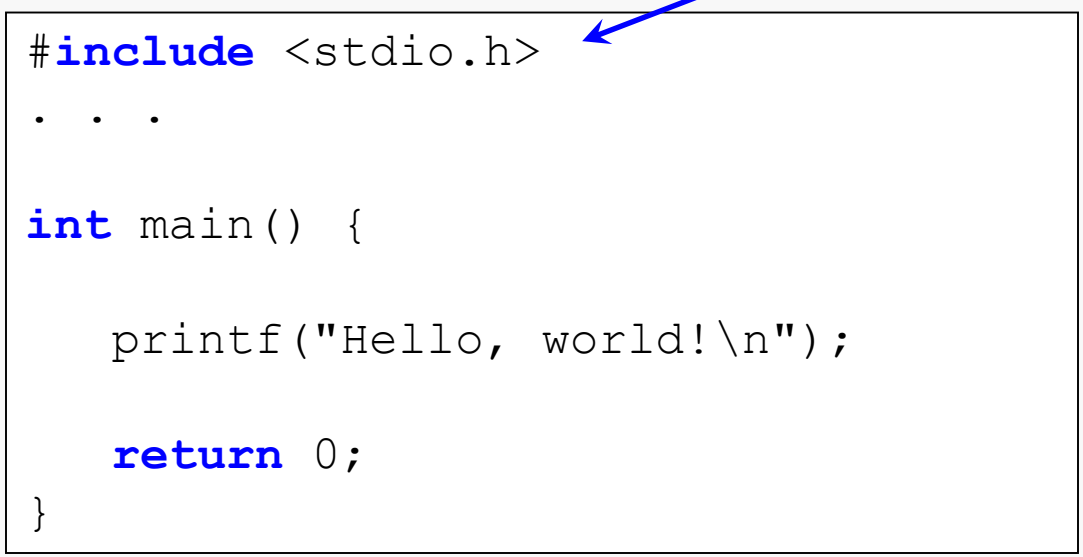
When a C compiler is invoked, the first thing that happens is that the code is parsed and modified by a *preprocessor*.

The preprocessor handles a collection of commands (commonly called *directives*), which are denoted by the character '#'.

`#include` directives specify an external file (for now a C library file); the preprocessor essentially copies the contents of the specified file in place of the directive.

We will see more interesting preprocessor directives later.

Contents of file `stdio.h` are copied here.



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

int main() {

   printf("Hello, world!\n");

   return 0;
}
```

The C Standard Library

The C Standard Library includes a fairly large collection of types and functions.

The declarations of these are placed into a collection of *header* files, which are part of the distribution of every C compiler.

The implementations are placed into a collection of C source files, which are then pre-compiled into binary *library* files (also part of every C compiler distribution).

C programmers incorporate portions of the Standard Library into their programs by making use of `#include` directives.

What's the Same as Java (more or less)

Naming rules are the same. But… customary conventions differ.

Declaration syntax is the same but semantics are different.

Scoping rules are similar, within a file at least.

Many reserved words are the same, with the same meanings, but ALL (almost) reserved words in C and ALL (almost) Standard Library identifiers are purely lower-case.

Operator symbols and expressions are generally the same.

The basic control structures (if, for, while, . . .) have same syntax and semantics.

Function call/return syntax and semantics are the same; as with Java, function parameters can only be passed into a function by value.

Conditionals and Loops

C includes the same set of conditional and loop statement forms as Java:

- `if…`
- `if…else…`
- `switch…`
- `while…`
- `for…`
- `do…while…`

C also has a `goto` statement for unconditional branching.

Thou shalt not `goto`.

C Philosophy

The stated goal of the designers of the C language is:

> Correct code should execute as fast as possible on the underlying hardware.

> Of course, good programmers write only correct code…

> … and only good programmers should be writing code.

Core Differences vs Java

All built-in C types are primitives; there are no class types in the language.

In C there is no notion of a member function.

A C program is a collection of functions that call one another, not a collection of classes and objects that use one another's services.

In C, every variable may be allocated dynamically, or not; it's up to you to decide.

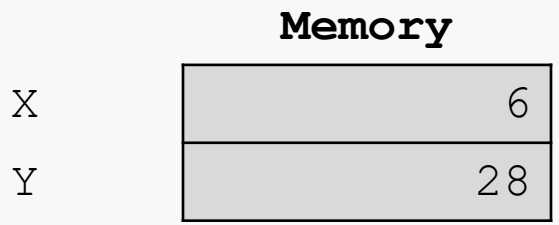
Scope rules are slightly different; a name declared within a block is strictly local to the block.

In most cases, C variables are not automatically initialized at all; you may initialize them yourself when you declare them. (Linux, however…)

Variable Declarations

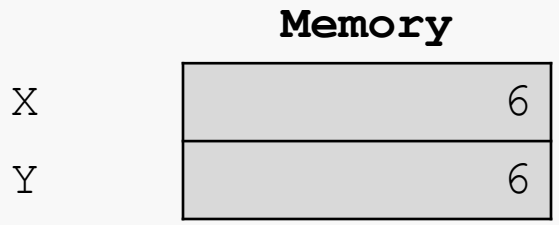
All declared objects are (by default) statically allocated (not dynamically). Thus, the following declaration results in `X` and `Y` being objects of type `int`, not references to objects:

```
int X =  6,
    Y = 28;
```



This has many consequences:

- assigning `X` to `Y` does not result in an alias; rather `X` becomes a copy of `Y`, but is still an entirely different object; just like Java primitives, and unlike Java objects
- using `X` as a parameter to a function does not allow the function to modify `X`
- logically, you can only initialize declared objects to 0 if they are numeric types

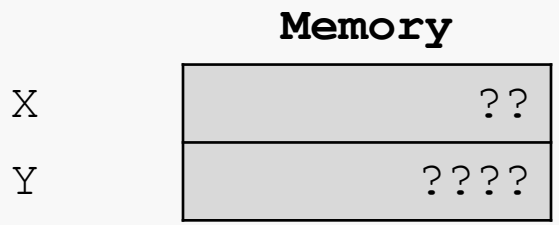


Automatic Variable Initialization

Variables are not (usually) automatically initialized.

The compiler will not check for use of a variable before it has been initialized.

```
int X, Y;
Y = 2*X + 1;
```



This is a common source of errors in C programs and is easily avoided.

Note: when Linux allocates memory to a process, it may write zeros into that memory, which has the effect of initializing variables stored within that memory to 0; you should never count on that to save you.

Boolean Variables

C initially did not have a Boolean type.

Integer values can be used as Booleans; zero is interpreted as false and all other values are interpreted as true.

Modern C includes a `_Bool` type which is aliased to `bool`.

Every expression in C has a value (well-defined or not). Hence, the following is valid code:

```
if ( x = 42 )
    // always executes the if-clause
```

C Primitive Types

Standard C provides a plethora of primitive types. These store single values, and are most definitely not objects in the Java sense. In particular, there is no guarantee of automatic initialization.

Integer types	Probable characteristics
`int`	32-bits
`unsigned int`	32-bits
`short (int)`	16-bits
`unsigned short (int)`	16-bits
`long (int)`	32-bits
`unsigned long (int)`	32-bits

```
#include <stdint.h>
int8_t       uint8_t
int16_t      uint16_t
int32_t      uint32_t
int64_t      uint64_t
```

Floating-point types	Conforming implementations provide:
`float`	32-bit IEEE single-precision type
`double`	64-bit IEEE double-precision type

C Primitive Types

Character types	Probable characteristics
`char`	1-byte, ASCII code
`unsigned char`	1-byte, unsigned integer

Logical types	Probable characteristics
`bool`	1-byte, value either `true` or `false` `<stdbool.h>` (really `_Bool`, but standard macro provides alias)

The primitive types, except as noted, are all available without any inclusions from the Standard Library.

C Arithmetic Operators

Same syntax as Java.

Semantics are generally the same as well, although the C Standard leaves the result of a number of unwise constructs undefined.

For example:

```
int x = 5;

x = x++ * x++;
```

Now, the C Standard leaves the result of executing that statement undefined. If you want a very detailed and interesting discussion of why this is so, take a look at:

`http://c-faq.com/~scs/readings/undef.950321.html`

My take on the issue is that such expressions are generally "stupid" and unlikely to be used in real code…

Precedence rules are the same as Java.

Precedence can be forced (and disambiguated) by use of parentheses.

Creating User-defined Types

Java

```
public class Rational {
  private int top;
  private int bottom;

  public Rational(...) {
    ...
  }
  ...
}
```

Classes:

- data and fn members
- member access control enforced by compiler
- automatic initialization (constructor must be invoked when object is created)

C

```
struct _Rational {
  int top;
  int bottom;
};
typedef struct _Rational Rational;

Rational Rational_Create(...) {
  ...
}
...
```

`struct` types:

- data members only
- member access control enforced by programmer discipline (or not)
- initialization only if programmer remembers to do it

Memory Management

Objects which are allocated dynamically are not automatically deallocated (at least, not until the program terminates execution).

Deallocating them efficiently is the responsibility of the programmer.

For now, we'll examine one simple case to illustrate the difference.

Java Object Creation


```
public class Rational {
  private int top;
  private int bottom;

  public Rational(...) {
    ...
  }
  ...
}
```

```
public void Calculate (...) {

  Rational r1 = new Rational(...);  // MUST alloc with new
  ...
  r1 = new Rational(...);           // Just discard old object
  ...
  // Dynamically-allocated objects are automatically reclaimed
  // (eventually) by the Java GC system... no worries!
}
```

C Object Creation I

```
struct _Rational {
  int top;
  int bottom;
};
typedef struct _Rational Rational;

Rational Rational_Create(...) {
  ...
}
...
```

```
void Calculate (...) {

  Rational r1 = Rational_Create(...);  // CAN create statically
  ...
  r1 = Rational_Create(...);           // Just discard old object
  ...
  // Statically-allocated objects are automatically reclaimed
  // when the function terminates... no worries!
}
```

Aside: Pointer Variables

A *pointer* is simply a variable whose value is the address of something.

When we allocate an object dynamically, we get an address:

```
Rational r1 = new Rational(...);
```

In the Java fragment above, `r1` is a *reference* variable, which is a kind of pointer, and the operator `new` returns the address of a chunk of memory which will hold the object being allocated.

In C, pointer variables are declared by using some "syntactic sugar" after the type specifier:

```
Rational* r1 = malloc(...);
```

The symbol `'*'` after the type specifier means that `r1` is a pointer.

The C function `malloc()` returns the address of a chunk of memory which will hold the object being allocated.

C Object Creation II

```
struct _Rational {
  int top;
  int bottom;
};
typedef struct _Rational Rational;

Rational Rational_Create(...) {
  ...
}
...
```

```
void Calculate (...) {

  Rational* r1 = malloc(...);   // CAN create dynamically
  ...
  free(r1);             // MUST explicitly destroy dynamic object
  r1 = malloc(...);     // before losing access to it
  ...
  // Dynamically-allocated objects are never automatically
  // reclaimed when fn terminates... worries!
}
```