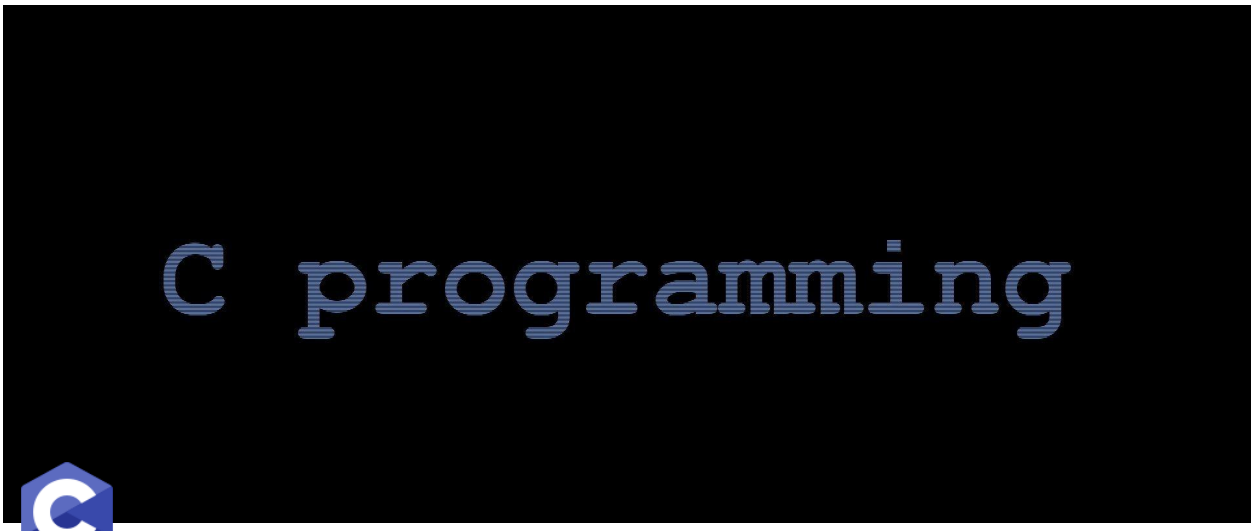


Week 1 (C)

- program life

  Source code >> Compiler >> Machine code

- ▼ First Program
  - To start a program

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

  - To add a function

```
int main(void) {

}
```

  - To print a string

```
printf("hello, world\n");
// use (\n) to start new line
```

- to print a variable

```
string name = "Hafez";
printf("Hello, %s\n");
```

▼ CS50 functions

1. get_char
2. get_double
3. get_float
4. get_int
5. get_long
6. get_string

▼ place Holders

- %c >>> char
- %f >>> float, double
- %i >>> int
- %li >>> long
- %s >>> string

▼ Variable

▼ Integers

▼ Create a Variable

```
int counte = 5;
```

▼ Modifing

```
// when modifing don't put type
counter ++ ;  // Adds 1
counter += 2; // Adds 2
```

▼ Using

- Using a variable
  - After a variable has been *declared*, it's no longer necessary to specify that variable's type. (In fact, doing so has some unintended consequences!)

```
int number;         // declaration
number = 17;        // assignment
char letter;        // declaration
letter = 'H';       // assignment
```

  - If you are simultaneously declaring and setting the value of a variable (sometimes called *initializing*), you can consolidate this to one step.

```
int number = 17;    // initialization
char letter = 'H'; // initialization
```

▼ Data Type

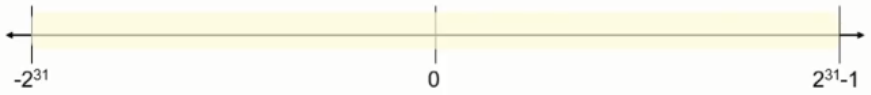
▼ int

```
#include <cs50.h>
#include <stdio.h>
```

```
int main(void){
    int age = get_int("What's Your age\n");
    int days = age * 360;
    printf("You are %i days old.\n", days);
}
```

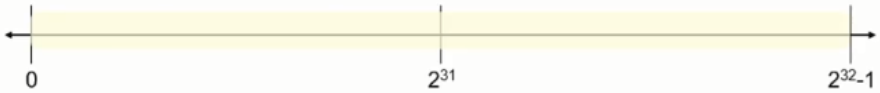
- `int`
  - The `int` data type is used for variables that will store integers.
  - Integers always take up 4 bytes of memory (32 bits). This means the range of values they can store is necessarily limited to 32 bits worth of information.

Integer Range



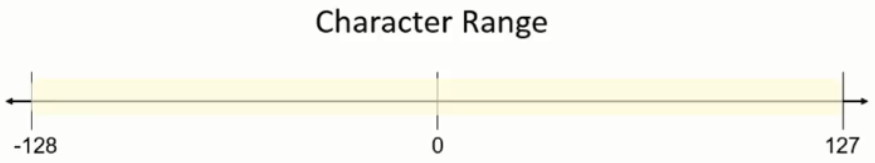
- `unsigned int`
  - `unsigned` is a *qualifier* that can be applied to certain types (including `int`), which effectively doubles the positive range of variables of that type, at the cost of disallowing any negative values.
  - You'll occasionally have use for `unsigned` variables in CS50.

Unsigned Integer Range



▼ `char`

- `char`
  - The `char` data type is used for variables that will store single characters.
  - Characters always take up 1 byte of memory (8 bits). This means the range of values they can store is necessarily limited to 8 bits worth of information.
  - Thanks to ASCII, we've developed a mapping of characters like A, B, C, etc... to numeric values in the positive side of this range.



▼ `float`

- `float`
  - The `float` data type is used for variables that will store floating-point values, also known as *real numbers*.
  - Floating points values always take up 4 bytes of memory (32 bits).
  - It's a little complicated to describe the range of a `float`, but suffice it to say with 32 bits of precision, some of which might be used for an integer part, we are limited in how *precise* we can be.

```
#include <cs50.h>
#include <stdio.h>

int main(void){
    float price = get_int("What's the Price \n");
    // %.2f >> to show only two degite
    printf("You pay %.2f$.\n", price * 1.25);
}
```

▼ `double >> float`

- `double`
  - The `double` data type is used for variables that will store floating-point values, also known as *real numbers*.
  - The difference is that doubles are *double precision.* They always take up 8 bytes of memory (64 bits).
  - With an additional 32 bits of precision relative to a `float`, `double`s allow us to be specify much more precise real numbers.

▼ `void`

- `void`
  - Is a type, but not a *data type*.
  - Functions can have a `void` return type, which just means they don't return a value.
  - The parameter list of a function can also be `void`. It simply means the function takes no parameters.
  - For now, think of `void` more as a placeholder for "nothing". It's more complex than that, but this should suffice for the better part of the course.

▼ `CS50`

- Those are the five primary types you'll encounter in C.
- In CS50, we also provide you with two additional types that will probably come in handy.

▼ `bool`

- `bool`
  - The `bool` data type is used for variables that will store a Boolean value. More precisely, they are capable only of storing one of two values: `true` and `false`.
  - Be sure to `#include <cs50.h>` atop your programs if you wish to use the `bool` type.

▼ `string`

- `string`
  - The `string` data type is used for variables that will store a series of characters, which programmers typically call a *string*.
  - Strings include things such as words, sentences, paragraphs, and the like.
  - Be sure to `#include <cs50.h>` atop your programs if you wish to use the `string` type.

▼ `Custom`

- Later in the course we'll also encounter structures (`structs`) and defined types (`typedefs`) that afford great flexibility in creating data types you need for your programs.

- Now, let's discuss how to create, manipulate, and otherwise work with variables using these data types.

▼ long >>

▼ Operators

▼ Arithmetic Operators

Arithmetic Operators

- In C we can add (`+`), subtract (`-`), multiply (`*`) and divide (`/`) numbers, as expected.

```
int x = y + 1;
x = x * 5;
```

- We also have the modulus operator, (`%`) which gives us the remainder when the number on the left of the operator is divided by the number on the right.

```
int m = 13 % 4; // m is now 1
```

Arithmetic Operators

- C also provides a shorthand way to apply an arithmetic operator to a single variable.

```
x = x * 5;
x *= 5;
```

- This trick works with all five basic arithmetic operators. C provides a further shorthand for incrementing or decrementing a variable by 1:

```
x++;
x--;
```

▼ Boolean Expirations

Boolean Expressions

- Boolean expressions are used in C for comparing values.
- All Boolean expressions in C evaluate to one of two possible values – `true` or `false`.
- We can use the result of evaluating a Boolean expression in other programming constructs such as deciding which branch in a *conditional* to take, or determining whether a *loop* should continue to run.

Boolean Expressions

- Sometimes when working with Boolean expressions we will use variables of type `bool`, but we don't have to.
- In C, *every* nonzero value is equivalent to `true`, and zero is `false`.
- Two main types of Boolean expressions: *logical operators* and *relational operators*.

- Logical operators
  - Logical AND (&&) is true if and only if both operands are true, otherwise false.

x	y	(x && y)
true	true	true
true	false	false
false	true	false
false	false	false

- Logical operators
  - Logical OR (||) is true if and only if at least one operand is true, otherwise false.

x	y	(x || y)
true	true	true
true	false	true
false	true	true
false	false	false

- Logical operators
  - Logical NOT (!) inverts the value of its operand.

x	!x
true	false
false	true

▼ Relational

- Relational operators
  - C also can test two variables for equality and inequality.
    - Equality (`x == y`)
    - Inequality (`x != y`)
  - Be careful! It's a common mistake to use the assignment operator (`=`) when you intend to use the equality operator (`==`).

- Relational operators
  - These behave as you would expect them to, and appear syntactically similar to how you may recall them from elementary arithmetic.
    - Less than (`x < y`)
    - Less than or equal to (`x <= y`)
    - Greater than (`x > y`)
    - Greater than or equal to (`x >= y`)

▼ Conditions

Conditionals

- Conditional expressions allow your programs to make decisions and take different forks in the road, depending on the values of variables or user input.

- C provides a few different ways to implement conditional expressions (also known as *branches*) in your programs, some of which likely look familiar from Scratch.

IF

Code

```
if (x < y)
{
printf("x is less than y\n");
}
else
{
printf("x is not less than y\n");
}
```

```
if (x < y)
{
  printf("x is less than y\n");
}
else if (x > y)
{
  printf("x is greater than y\n");
}
else
{
  printf("x is equal to y\n");
}
```

```c
#include <cs50.h>
#include <stdio.h>

int main(void){
    int n = get_int("n: ");
    if(n % 2 == 0){
        printf("(%i) is even\n",n);
    }else{
        printf("(%i) is odd\n",n);
    }
}
```

▼ slides

```
if (boolean-expression)
{

}
```

- If the boolean-expression evaluates to true, all lines of code between the curly braces will execute in order from top-to-bottom.
- If the boolean-expression evaluates to false, those lines of code will not execute.

```
if (boolean-expression)
{

}
else
{

}
```

- If the `boolean-expression` evaluates to `true`, all lines of code between the first set of curly braces will execute in order from top-to-bottom.
- If the `boolean-expression` evaluates to `false`, all lines of code between the second set of curly braces will execute in order from top-to-bottom.

```
if (boolean-expr1)
{
   // first branch
}
else if (boolean-expr2)
{
   // second branch
}
else if (boolean-expr3)
{
   // third branch
}
else
{
   // fourth branch
}
```

- In C, it is possible to create an `if-else if-else` chain.
  - In Scratch, this required nesting blocks.
- As you would expect, each branch is mutually exclusive.

```
if (boolean-expr1)
{
   // first branch
}
if (boolean-expr2)
{
   // second branch
}
if (boolean-expr3)
{
   // third branch
}
else
{
   // fourth branch
}
```

- It is also possible to create a chain of non-mutually exclusive branches.
- In this example, only the third and fourth branches are mutually exclusive. The `else` binds to the nearest `if` only.

▼ Switch

▼ slides

```
int x = GetInt();
switch(x)
{
   case 1:
      printf("One!\n");
      break;
   case 2:
      printf("Two!\n");
      break;
   case 3:
      printf("Three!\n");
      break;
   default:
      printf("Sorry!\n");
}
```

- C's `switch()` statement is a conditional statement that permits enumeration of discrete cases, instead of relying on Boolean expressions.
- It's important to `break;` between each case, or you will "fall through" each case (unless that is desired behavior).

```c
int x = GetInt();
switch(x)
{
   case 5:
      printf("Five, ");
   case 4:
      printf("Four, ");
   case 3:
      printf("Three, ");
   case 2:
      printf("Two, ");
   case 1:
      printf("One, ");
   default:
      printf("Blast-
              off!\n");
}
```

- C's `switch()` statement is a conditional statement that permits enumeration of discrete cases, instead of relying on Boolean expressions.
- It's important to `break;` between each case, or you will "fall through" each case (unless that is desired behavior).

▼ Ternary

▼ slides

```c
int x;
if (expr)
{
   x = 5;
}
else
{
   x = 6;
}
```

```c
int x = (expr) ? 5 : 6;
```

- These two snippets of code act identically.
- The ternary operator (`?:`) is mostly a cute trick, but is useful for writing trivially short conditional branches. Be familiar with it, but know that you won't need to write it if you don't want to.

```c
int x;
if (expr)
{
   x = 5;
}
else
{
   x = 6;
}
```

```c
int x = (expr) ? 5 : 6;
```

- These two snippets of code act identically.
- The ternary operator (`?:`) is mostly a cute trick, but is useful for writing trivially short conditional branches. Be familiar with it, but know that you won't need to write it if you don't want to.

`if` (and `if-else`, and `if-else if-…-else`)

- Use Boolean expressions to make decisions.

`switch`

- Use discrete cases to make decisions.

`?:`

- Use to replace a very simple `if-else` to make your code look fancy.

▼ Loops

- Loops allow your programs to execute lines of code repeatedly, saving you from needing to copy and paste or otherwise repeat lines of code.

- C provides a few different ways to implement loops in your programs, some of which likely look familiar from Scratch.

▼ While Loop

```
// A loop runs forever
while (true)
{
  printf("hello, world\n");
}
```

```
int i = 0;
while (i < 5for (int i = 0; i< 5; i++){
        printf("%i.True is True\n", i);
    }){
    printf("%i.True is True\n", i);
    i++;
}
```

```
while (true)
{

}
```

- This is what we call an *infinite loop*. The lines of code between the curly braces will execute repeatedly from top to bottom, until and unless we break out of it (as with a `break;` statement) or otherwise kill our program.

```
while (boolean-expr)
{

}
```

- If the `boolean-expr` evaluates to `true`, all lines of code between the curly braces will execute repeatedly, in order from top-to-bottom, until `boolean-expr` evaluates to `false`.
- Somewhat confusingly, the behavior of the Scratch block is reversed, but it is the closest analog.

▼ For Loop

```
for (int i = 0; i< 5; i++) {
  printf("%i.True is True\n", i);
}
```

```
for (int i = 0; i < 10; i++)
{

}
```

- Syntactically unattractive, but `for` loops are used to repeat the body of a loop a specified number of times, in this example 10.
- The process undertaken in a `for` loop is:
  - The counter variable(s) (here, `i`) is set
  - The Boolean expression is checked.
    - If it evaluates to `true`, the body of the loop executes.
    - If it evaluates to `false`, the body of the loop does not execute.
  - The counter variable is incremented, and then the Boolean expression is checked again, etc.

```
for (start; expr; increment)
{

}
```

- Syntactically unattractive, but `for` loops are used to repeat the body of a loop a specified number of times, in this example 10.
- The process undertaken in a `for` loop is:
  - The statement(s) in `start` are executed
  - The `expr` is checked.
    - If it evaluates to `true`, the body of the loop executes.
    - If it evaluates to `false`, the body of the loop does not execute.
  - The statement(s) in `increment` are executed, and then the `expr` is checked again, etc.

▼ Do while

```c
#include <cs50.h>
#include <stdio.h>

void hello(int n);

int main(void)
{
    int n = 0;
    do{
        printf("%i: hello\n",n);
        n++;
    }
    while(n < 9);
}
```

```
do
{

}
while (boolean-expr);
```

- This loop will execute all lines of code between the curly braces once, and then, if the `boolean-expr` evaluates to `true`, will go back and repeat that process until `boolean-expr` evaluates to `false`.

`while`

- Use when you want a loop to repeat an unknown number of times, and possibly not at all.

`do-while`

- Use when you want a loop to repeat an unknown number of times, but at least once.

`for`

- Use when you want a loop to repeat a discrete number of times, though you may not know the number at the moment the program is compiled.