

BASICS OF C

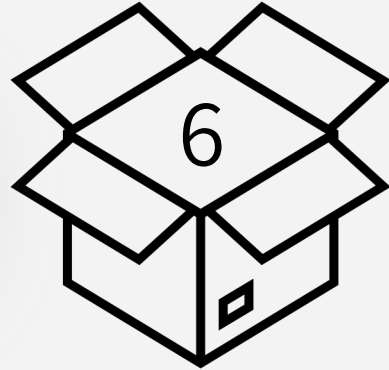
VARIABLES AND DATA TYPES

What is a variable?

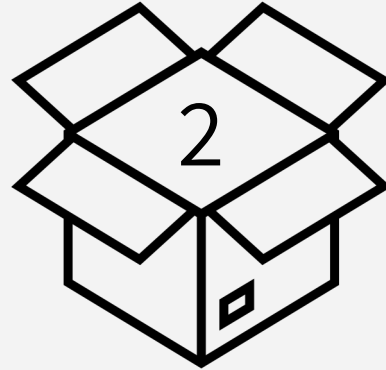
To write programs that perform useful tasks that really save us work, we need to introduce the concept of **variables**.

➤ 5+1
➤ 2 } subtract

*FROM PREVIOUS
LECTURE!*



a



b



result

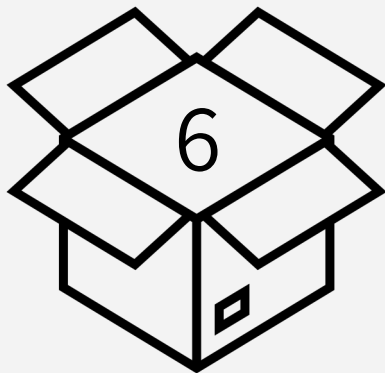
What is a variable?

To write programs that perform useful tasks that really save us work, we need to introduce the concept of **variables**.

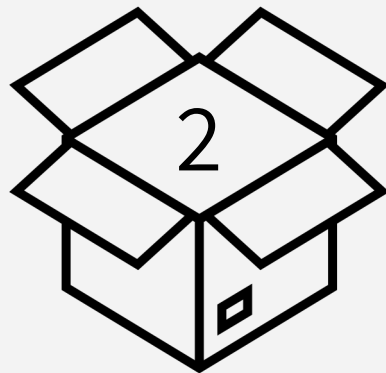
➤ 5+1
➤ 2 } subtract

almost C code

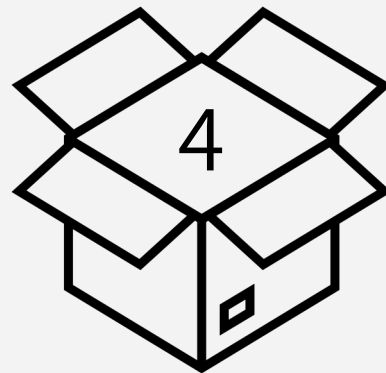
```
a = 5;  
b = 2;  
a = a + 1;  
result = a - b;
```



a



b



result

Identifiers

An identifier:

- can be a sequence of one or more letters, digits, or underscore characters (_)
- cannot contain spaces, punctuation marks, and symbols
- shall always begin with a letter or the underscore character (_)

Standard reserved keywords that cannot be used to create identifiers:

```
auto, break, case, char, const, continue, default, do, double, else, enum,  
extern, float, for, goto, if, int, long, register, return, short, signed,  
sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while
```

Identifiers

C is case sensitive.

`result` \neq `RESULT` \neq `rEsUlT`

Properties of C data types

➤ C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

```
a = 5;  
b = 2;  
result = a - b;  
c = reuslt * 3;
```

Identifier	Value


Properties of C data types

➤ C is strongly typed.

variables are bound to specific
data types

data types were not specified

WEAKLY-TYPED LANGUAGE



```
a = 5;  
b = 2;  
result = a - b;  
c = reuslt * 3;
```

Identifier	Value

Properties of C data types

➤ C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

➤ `a = 5;`
`b = 2;`
`result = a - b;`
`c = reuslt * 3;`

Identifier	Value
a	5

Properties of C data types

➤ C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

➤
`a = 5;`
`b = 2;`
`result = a - b;`
`c = reuslt * 3;`

Identifier	Value
a	5
b	2

Properties of C data types

➤ C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

```
a = 5;  
b = 2;  
➤ result = a - b;  
c = reuslt * 3;
```

Identifier	Value
a	5
b	2
result	3

Properties of C data types

➤ C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

```
a = 5;  
b = 2;  
result = a - b;  
➤ c = reuslt * 3; 3*3=9
```

Identifier	Value
a	5
b	2
result	3

Properties of C data types

- C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

```
a = 5;  
b = 2;  
result = a - b;  
➤ c = reuslt * 3; 3*3=9
```

↑ typo!!!!

Identifier	Value
a	5
b	2
result	3

Properties of C data types

- C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

```
a = 5;  
b = 2;  
result = a - b;  
➤ c = reuslt * 3;
```

↑ typo!!!!

Identifier	Value
a	5
b	2
result	3
reuslt	0
c	

Properties of C data types

- C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

```
a = 5;  
b = 2;  
result = a - b;  
➤ c = reuslt * 3;
```

↑ typo!!!!

Identifier	Value
a	5
b	2
result	3
reuslt	0
c	0

Properties of C data types

- C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

```
a = 5;  
b = 2;  
result = a - b;  
➤ c = reuslt * 3;
```

↑ typo!!!!

Identifier	Value
a	5
b	2
result	3
reuslt	0
c	0

with C we would have a compile error!

Properties of C data types

- C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

```
a = 5;  
b = 2;  
result = a - b;  
➤ c = reuslt * 3;  
      ↑  
      typo!!!!
```

- Storing an integer is different than storing a floating-point number.

158 \neq 1.58

Identifier	Value
a	5
b	2
result	3
reuslt	0
c	0

with C we would have a compile error!

Properties of C data types

- C is strongly typed.

variables are bound to specific
data types

WEAKLY-TYPED LANGUAGE

```
a = 5;  
b = 2;  
result = a - b;  
➤ c = reuslt * 3;  
      ↑  
      typo!!!!
```

Identifier	Value
a	5
b	2
result	3
reuslt	0
c	0

- Storing an integer is different than storing a floating-point number.

158 \neq 1.58

with C we would have a compile error!

- Different types occupy different amount of memory.



Fundamental data types

Fundamental data types are basic types implemented directly by the language that represent the basic storage units supported natively by most systems:

- **Character types:** a single character, such as **'A'** or **'\$'**.
- **Numerical integer types:** numeric values, such as **7** or **1024**, can either be signed or unsigned.
- **Floating-point types:** real values, such as **3.14** or **0.01**, with different levels of precision, depending on which of the three floating-point types is used.
- **Boolean type:** either **true** or **false**

Fundamental data types

almost C code

```
a = 5;  
b = 2;  
a = a + 1;  
result = a - b;
```



C code

```
int a = 5;  
int b = 2;  
a = a + 1;  
int result = a - b;
```

Fundamental data types

Character types

Type names	Notes on size / precision
char	Exactly one byte in size. At least 8 bits.
char16_t	Not smaller than char . At least 16 bits.
char32_t	Not smaller than char16_t . At least 32 bits.
wchar_t	Can represent the largest supported character set.

Floating-points types

float	Same size as int . At least 8 bits.
double	Precision not less than float .
long double	Precision not less than double .

Other types

bool	Exactly one byte in size.
void	No storage.
decltype (nullptr)	No storage.

Integer types (signed)

Type names	Notes on size / precision
signed char	Same size as char . At least 8 bits.
<i>signed short int</i>	Not smaller than char . At least 16 bits.
<i>signed int</i>	Not smaller than short . At least 16 bits.
<i>signed long int</i>	Not smaller than int . At least 32 bits.
<i>signed long long int</i>	Not smaller than long . At least 64 bits.

Integer types (unsigned)

unsigned char	Same size as char . At least 8 bits.
unsigned short int	Not smaller than char . At least 16 bits.
unsigned int	Not smaller than short . At least 16 bits.
unsigned long int	Not smaller than int . At least 32 bits.
unsigned long long int	Not smaller than long . At least 64 bits.

Type representable values

Type sizes are expressed in bits (or bytes); the more bits a type has, the more distinct values it can represent, but at the same time, also consumes more space in memory:

1 byte
=
8 bits

Size	Unique representable values	Notes
8 bit	256	= 2^8
16 bit	65,536	= 2^{16}
32 bit	4,294,967,296	= 2^{32}
64 bit	18,446,744,073,709,551,616	= 2^{64}

#combinations
=
symbols^{digits}

Declaration of variables

C is **strongly typed**: it requires every variable to be declared with its type before its first use. This informs the compiler the size to reserve in memory for the variable and how to interpret its value.

type name;

type of the variable

identifier

Declaration of variables

C is **strongly typed**: it requires every variable to be declared with its type before its first use. This informs the compiler the size to reserve in memory for the variable and how to interpret its value.

type name;
↑ ↙
type of the variable identifier

```
int x;  
float y;  
double z;  
char c;
```

```
int a;  
int b;  
int c;
```

same as

```
int a, b, c;
```


Initialization of variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time.

```
int x;  
float y;  
double z;  
char c;  
  
printf("int = %d\n", x);  
printf("float = %f\n", y);  
printf("double = %lf\n", z);  
printf("char = %c\n", c);
```

Initialization of variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time.

```
int x;  
float y;  
double z;  
char c;
```

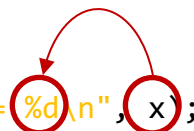
```
printf("int = %d\n", x);  
printf("float = %f\n", y);  
printf("double = %lf\n", z);  
printf("char = %c\n", c);
```

These special characters will associate the variable we want to print to their location in the message and their type. That's how **printf** works.

Initialization of variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time.

```
int x;  
float y;  
double z;  
char c;  
  
printf("int = %d\n", x);  
printf("float = %f\n", y);  
printf("double = %lf\n", z);  
printf("char = %c\n", c);
```

A diagram with two red circles. The left circle is around the format specifier '%d' in the first printf statement. The right circle is around the variable 'x' in the same statement. A red curved arrow points from the right circle to the left circle, indicating the association between the variable and its format specifier.

These special characters will associate the variable we want to print to their location in the message and their type. That's how **printf** works.

For example, **%d** will be associated to the variable **x**. When printing the message, the value of **x** will be printed instead of **%d**.

Initialization of variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time.

```
int x;  
float y;  
double z;  
char c;
```

```
printf("int = %d\n", x);  
printf("float = %f\n", y);  
printf("double = %lf\n", z);  
printf("char = %c\n", c);
```

These special characters will associate the variable we want to print to their location in the message and their type. That's how **printf** works.

For example, **%d** will be associated to the variable **x**. When printing the message, the value of **x** will be printed instead of **%d**.

%d indicates that we want to print an **int**.

%f indicates that we want to print a **float**.

%lf indicates that we want to print a **double**.

%c indicates that we want to print a **char**.

We will see more details at the end of this lecture.

Initialization of variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time.

```
int x;  
float y;  
double z;  
char c;
```

```
➤ printf("int = %d\n", x);  
printf("float = %f\n", y);  
printf("double = %lf\n", z);  
printf("char = %c\n", c);
```

```
int = -1012919088
```

Initialization of variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time.

```
int x;  
float y;  
double z;  
char c;  
  
printf("int = %d\n", x);  
➤ printf("float = %f\n", y);  
printf("double = %lf\n", z);  
printf("char = %c\n", c);
```

```
int = -1012919088  
float = 0.000000
```

Initialization of variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time.

```
int x;  
float y;  
double z;  
char c;  
  
printf("int = %d\n", x);  
printf("float = %f\n", y);  
➤ printf("double = %lf\n", z);  
printf("char = %c\n", c);
```

```
int = -1012919088  
float = 0.000000  
double = 0.000000
```

Initialization of variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time.

```
int x;  
float y;  
double z;  
char c;  
  
printf("int = %d\n", x);  
printf("float = %f\n", y);  
printf("double = %lf\n", z);  
➤ printf("char = %c\n", c);
```

```
int = -1012919088  
float = 0.000000  
double = 0.000000  
char =
```


Initialization of variables

It is possible for a variable to have a specific value from the moment it is declared. This is called the initialization of the variable.

```
int x = 10;
float y = 15.4f;
double z = 20.2;
char c = 'a';

printf("int = %d\n", x);
printf("float = %f\n", y);
printf("double = %lf\n", z);
printf("char = %c\n", c);
```

type name = value;

```
int = 10
float = 15.4
double = 20.2
char = a
```

CONSTANTS

Constants are expressions with a fixed value.

Literals

```
a = 5;
```

The **5** in this piece of code is a *literal constant*. **a** is the variable that takes its value!

Literal constants can be classified into:

- integer,
- floating-point,
- characters,
- Boolean,
- pointers, and
- user-defined literals.

Literals: integer numbers

```
1000
45
-213
```

A simple succession of digits representing a whole number in decimal base.

```
75          // 75 decimal (base 10)
0b1001011  // 75 binary (base 2), digits are preceded by '0b'
0113       // 75 octal (base 8), digits are preceded by '0'
0x4b       // 75 hexadecimal (base 16), digits are preceded by '0x'
```

C allows number representation in different bases.

```
75    // int
75u   // unsigned int
75U   // unsigned int
75l   // long
75L   // long
75ul  // unsigned long
75lu  // unsigned long
75ll  // long long
75ull // unsigned long long
```

Literals have **int** type by default.

Suffix	Type modifier
u or U	unsigned
l or L	long
ll or LL	long long

Literals: floating-point numbers

```
3.0      // 3.0
3.1459   // 3.1459
6.02e3    // 6.02 x 10^3
1.6E-19   // 1.6 x 10^-19
```

Can include either:

- a decimal point,
- an **e** or **E** character (that expresses "by ten at the Xth height", where X is an integer value that follows the **e** character), or
- both a decimal point and an **e** character.

The default type for floating-point literals is **double**.

```
3.1459    // double
3.1459f    // float
3.1459l    // long double
```

Suffix	Type modifier
f or F	float
l or L	long double

Literals: characters

```
'z'      // character  
'p'      // character
```

Character literals are enclosed in single quotes.

```
x        // identifier of a variable  
'x'      // character literal
```

Difference between identifiers and literals.

Escape code	Description
<code>\n</code>	newline
<code>\t</code>	tab
<code>\a</code>	alert (plays a sound)
<code>\'</code>	single quote
<code>\"</code>	double quote

Can also represent special characters that are difficult or impossible to express otherwise in the source code of a program.

Escape code	Description
<code>\?</code>	question mark
<code>\\</code>	backslash

Literals: characters and strings

```
1: #include <stdio.h>
2:
3: int main()
4: {
5:     printf("Numbers:\t%d\t%d\t%d", 1, 2, 3);
6: }
```

Numbers: 1 2 3

Literals: characters and strings

```
1: #include <stdio.h>
2:
3: int main()
4: {
5:     printf("Numbers:\t%d\t%d\t%d", 1, 2, 3);
6: }
```

tabulation

Numbers:  1  2  3

Preprocessor definitions (#define)

Sometimes, it is just convenient to give a name to a constant value.

```
1: #include <stdio.h>
2:
3: #define PI 3.14159
4: #define NEWLINE '\n'
5:
6: int main()
7: {
8:     double r = 5.0;
9:     double circle = 2.0 * PI * r;
10:    printf("Radius: %lf%c", r, NEWLINE);
11:    printf(" Circle: %lf";
12: }
```

← #define identifier replacement

#define lines are preprocessor directives, and as such are single-line instructions that -unlike C statements- do not require semicolons ; at the end

```
Radius: 5.0
Circle: 31.4159
```

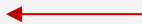
OPERATORS

Assignment operator

The assignment operator `=` assigns a value to a variable.

```
int a = 5;           // the value of a is 5
int b = a;           // the value of b is a, which is 5
int c = 2 + a;        // the value of c is 2 + a, which is 7
a = b = c;           // the value of a and b is c, which is 7
```

direction of assignment




identifier = value

Arithmetic operators

The five arithmetical operations supported by C are:

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
%	modulo

```
x = 5 + 5;    // x = 10
x = 5 - 4;    // x = 1
x = 5 * 5;    // x = 25
x = 5 / 5;    // x = 1
x = 11 % 3;   // x = 2
```



modulo gives the
remainder of a division of
two values

Compound assignment

Compound assignment operators modify the current value of a variable by performing an operation on it.

Expression	Equivalent to...
<code>y+=x;</code>	<code>y = y + x;</code>
<code>y-=x;</code>	<code>y = y - x;</code>
<code>y/=x;</code>	<code>y = y / x;</code>
<code>y*=(x+1);</code>	<code>y = y * (x + 1);</code>

Increment and decrement

Increase or reduce by one the value stored in a variable.

```
x = x + 1;  
x += 1;  
// are equivalent to  
x++;  
++x;
```

```
x = x - 1;  
x -= 1;  
// are equivalent to  
x--;  
--x;
```

- As a prefix (**++X**): first increases, then uses the value in the expression
- As a postfix or suffix (**X++**) first uses the value in the expression, then increases

```
// prefix example  
x = 1;  
y = 5 + (++x);  
// x = 2 , y = 7
```

```
// suffix example  
x = 1;  
y = 5 + (x++);  
// x = 2 , y = 6
```

Relational and comparison operators

Two expressions can be compared using relational and equality operators, resulting in either **true** or **false**.

Operator	Description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

```
(7 == 5)    // evaluates to false
(7 != 5)    // evaluates to true
(7 < 5)     // evaluates to false
(7 > 5)     // evaluates to true
(7 <= 5)    // evaluates to false
(7 >= 5)    // evaluates to true
```

```
int a = 7, b = 5;
```

```
(a == b)    // evaluates to false
(a != b)    // evaluates to true
(a < b)     // evaluates to false
(a > b)     // evaluates to true
(a <= b)    // evaluates to false
(a >= b)    // evaluates to true
```

Relational and comparison operators

Two expressions can be compared using relational and equality operators, resulting in either **true** or **false**.

Operator	Description
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal to
<code>>=</code>	Greater than or equal to

```
(7 == 5)    // evaluates to false
(7 != 5)    // evaluates to true
(7 < 5)     // evaluates to false
(7 > 5)     // evaluates to true
(7 <= 5)    // evaluates to false
(7 >= 5)    // evaluates to true
```

```
int a = 7, b = 5;
```

```
(a == b)    // evaluates to false
(a != b)    // evaluates to true
(a < b)     // evaluates to false
(a > b)     // evaluates to true
(a <= b)    // evaluates to false
(a >= b)    // evaluates to true
```

BE CAREFUL:
= is not ==

Logical operators

Logical operators allow for Boolean expressions.

Operator	Description
!	Not
&&	And
	Or

```
!(5 == 5)           // evaluates to not true → false
!false              // evaluates to not false → true
((5 == 5) && (3 > 6)) // evaluates to true AND false → false
((5 == 5) || (3 > 6)) // evaluates to true OR false → true
```

Operator	Short-circuit
&&	If the left-hand side expression is false , the combined result is false (the right-hand side expression is never evaluated).
	If the left-hand side expression is true , the combined result is true (the right-hand side expression is never evaluated).

`((5 == 5) || (3 > 6))`



Once C checked that (5==5) is true, it will never check for the value of (3>6)

Logical operators – Checking Ranges

THIS IS NOT C
1 < x < 10

THIS IS C
1 < x && x < 10

Conditional ternary operator

The conditional operator evaluates an expression, returning one value if that expression evaluates to **true**, and a different one if the expression evaluates as **false**.

```
condition ? result if true : result if false
```

```
7 == 5 ? 4 : 3 // evaluates to 3
```

Explicit type casting operator

Type casting operators allow you to convert a value of a given type to another type.

```
int i;  
float f = 3.14f;  
i = (int) f;           // i = 3
```

variable = (type) formula

*BE CAREFUL TO
THE TYPES IN
YOUR
FORMULAS!*

$$\frac{124}{3} = 41.\bar{3}$$

```
int x = 124, y = 3;  
printf("%d", x / y);
```

41

Explicit type casting operator

Type casting operators allow you to convert a value of a given type to another type.

```
int i;  
float f = 3.14f;  
i = (int) f;           // i = 3
```

variable = (type) formula

*BE CAREFUL TO
THE TYPES IN
YOUR
FORMULAS!*

$$\frac{124}{3} = 41.\bar{3}$$

```
int x = 124, y = 3;  
printf("%f", (float)x / (float)y);
```

41.3333

Precedence rules

A single expression may have multiple operators. For example:

```
x = 5 + 7 % 2;    // x = 5 + 1 = 6, because % has higher precedence than +  
x = 5 + (7 % 2);  // x = 5 + 1 = 6, makes explicitly clear the intended effect  
x = (5 + 7) % 2;  // x = 12 % 2 = 0, overrides the precedence order
```

*USE PARENTHESES
IN YOUR
EXPRESSIONS*

*SUFFIX/POSTFIX
HAVE
PRECEDENCE
OVER PREFIX*

BASIC INPUT/OUTPUT

Standard output (printf)

The standard output by default is the screen (i.e., the monitor).

```
printf("int = %d\n", x);           // print the value of x, which is an int
```


Standard output (printf)

The standard output by default is the screen (i.e., the monitor).

```
printf("int = %d\n", x);           // print the value of x, which is an int
```



Message to print

Standard output (printf)

The standard output by default is the screen (i.e., the monitor).

```
printf("int = %d\n", x);    // print the value of x, which is an int
```



Format specifier

Standard output (printf)

The standard output by default is the screen (i.e., the monitor).

```
printf("int = %d\n", x);    // print the value of x, which is an int
```

Format specifier

Variable to print

Standard output (printf)

The standard output by default is the screen (i.e., the monitor).

```
printf("int = %d\n", x);    // print the value of x, which is an int
```

A red circle highlights the format specifier "%d" in the printf statement. Another red circle highlights the variable "x". A red curved arrow points from the circle around "x" to the circle around "%d", indicating that the value of x is passed to the %d format specifier.

Associates the value of **x** to that portion of the message.

Standard output (printf)

The standard output by default is the screen (i.e., the monitor).

```
printf("int = %d\n", x);    // print the value of x, which is an int
```

Format Specifier	Type
%c	char
%d	int
%f	float
%hi	short
%hu	unsigned short
%i or %u	unsigned int
%l or %ld	long

Format Specifier	Type
%lf	double
%Lf	long double
%lu	unsigned long
%lli or %lld	long long
%llu	unsigned long long
%s	string

Standard input (scanf)

The standard output by default is the keyboard.

```
int age;  
printf("Enter your age: ");  
scanf("%d", &age);  
// print the message "Enter your age:"  
// assign to 'age' the value received from the keyboard
```

Standard input (scanf)

The standard output by default is the keyboard.

```
int age;  
printf("Enter your age: "); // print the message "Enter your age:"  
scanf("%d", &age);        // assign to 'age' the value received from the keyboard
```



Format specifier
Like in **printf**

Standard input (scanf)

The standard output by default is the keyboard.

```
int age;  
printf("Enter your age: ");    // print the message "Enter your age:"  
scanf("%d", &age);           // assign to 'age' the value received from the keyboard
```


Format specifier
Like in **printf**

Variable to be assigned
the value by keyboard

Standard input (scanf)

The standard output by default is the keyboard.

```
int age;  
printf("Enter your age: ");    // print the message "Enter your age:"  
scanf("%d", &age);           // assign to 'age' the value received from the keyboard
```



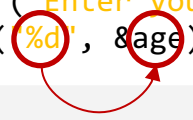
This operator is called address-of operator, and it returns the address in memory of the associated variable (i.e., where this variable is saved in RAM).

We will go more into details about this operator in CE 241 - Computer Programming II; for now, just accept it as it is.

Standard input (scanf)

The standard output by default is the keyboard.

```
int age;  
printf("Enter your age: ");  
scanf("%d", &age);  
  
// print the message "Enter your age:"  
// assign to 'age' the value received from the keyboard
```

A diagram consisting of two red circles. The first circle is around the format specifier "%d" in the scanf function call. The second circle is around the variable "age" in the same function call. A red curved arrow points from the first circle to the second circle, indicating that the value read from the keyboard is stored in the variable 'age'.

Stores the value received by
keyboard in the variable **age**.

Standard input (scanf)

The standard output by default is the keyboard.

```
int age;  
printf("Enter your age: ");    // print the message "Enter your age:"  
scanf("%d", &age);           // assign to 'age' the value received from the keyboard
```

This operation makes the program wait for input. The characters introduced using the keyboard are only transmitted to the program when the ENTER (or spaces such as whitespaces, tabs, new-line...) key is pressed.

Standard input (scanf)

The standard output by default is the keyboard.

```
int age;  
printf("Enter your age: ");           // print the message "Enter your age:"  
scanf("%d", &age);                   // assign to 'age' the value received from the keyboard
```

This operation makes the program wait for input. The characters introduced using the keyboard are only transmitted to the program when the ENTER (or spaces such as whitespaces, tabs, new-line...) key is pressed.

Type cast depends on the format specifier..