Expressions and Functions in C

Data types and operators – recap

Basic data types in C:

Signed integers:	Unsigned integers:	Floating point numbers:
int	**char**	float
long (int)	unsigned (int)	**double**
long long (int)	unsigned long (int)	long double
	unsigned long long (int)	

Basic operators in C:

- Bitwise `&, |, ~, ^, << , >>`
- Relational and logical `>, >=, <, <=, ==, !=, &&, ||`
- Arithmetic `+, –, *, /, %`
- Assignment `=, +=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=`
- Increment / decrement `++, ––`

Operator complexity – summary

- Relational operators (e.g., `>`) involve simple comparisons that can be done in linear time or less
- Logical operators (e.g., `&&`) involve comparing a series of integers to (linear or less)
- Bitwise operators (e.g., `|`) typically involve linear scans of the bit vector
- Integer arithmetic operators are linear (e.g., +,-) or quadratic (*,/)
- Floating point arithmetic operations are more complex

Multiple operators

- General question: how do we parse expressions that involve multiple operators?

```
int a = 3 / 4.0;
```

```
y = ++x;
```

```
a = c<d << 3;
```

```
float f = 3/4;
```

```
printf ("%d\n", a<a<<2 && a-7);
```

```
if(a<c || --b); vs. if(a < c||--b);
```

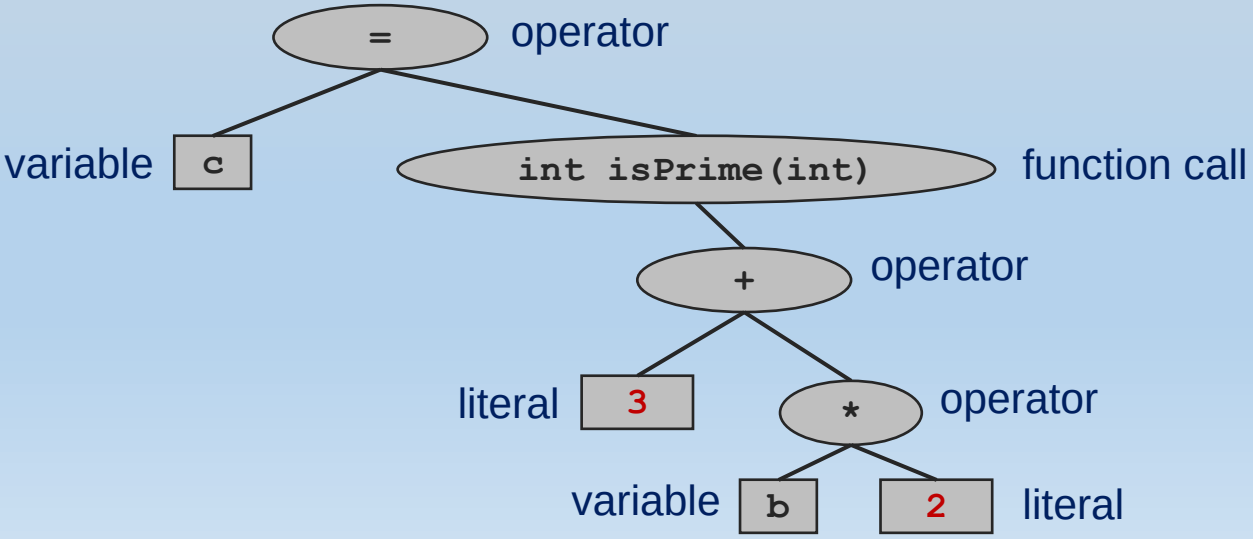
```
int a = b -= 2;
```

Lecture overview

- Expression trees ←
- Type conversion
- Function definition and invocation
- Function declaration and prototypes

Expression trees

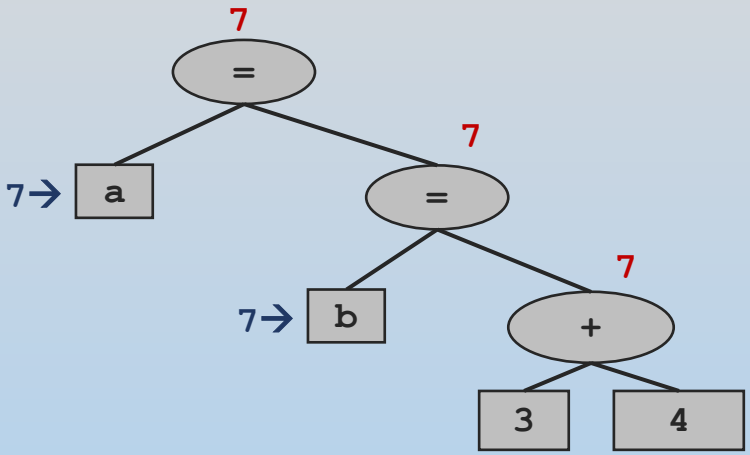
- Expressions can be described using a tree structure
- Leaves: variables or literals (explicit values)
- Internal vertices: operators or function calls
- Evaluation of expression is done bottom-up
- Example: `int c = isPrime(3 + b * 2)`



Expression trees

Example 1:

```
int a = b = (3+4);
```



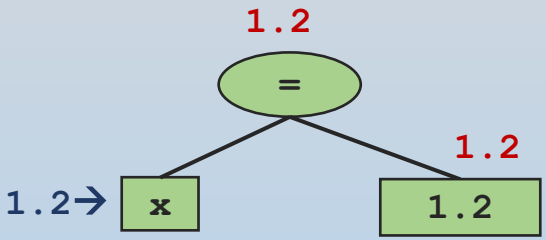
- Assignment operators have side effects (they change variable value) but they also **return a value**

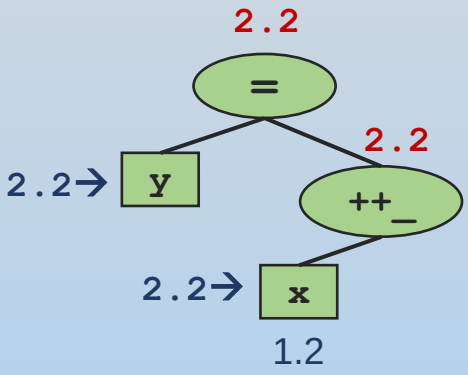
Expression trees

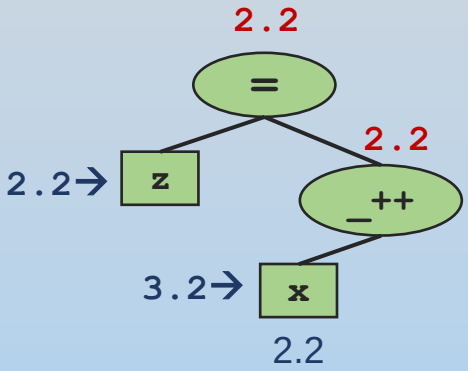
Example 2:

```
double x = 1.2;
double y = ++x;
double z = x++;
```

← y=?
← z=?





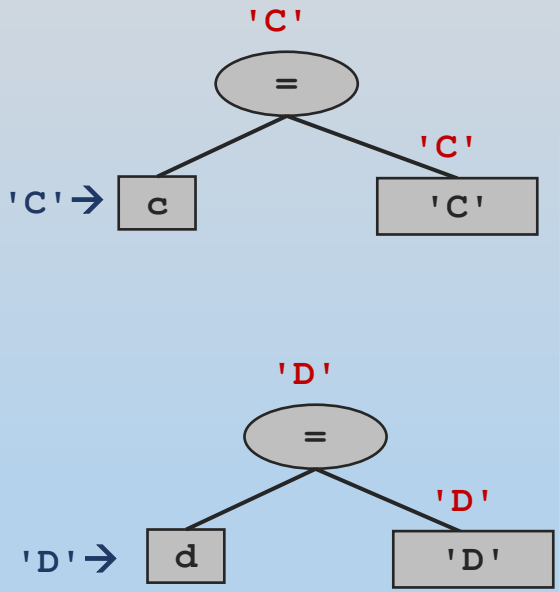


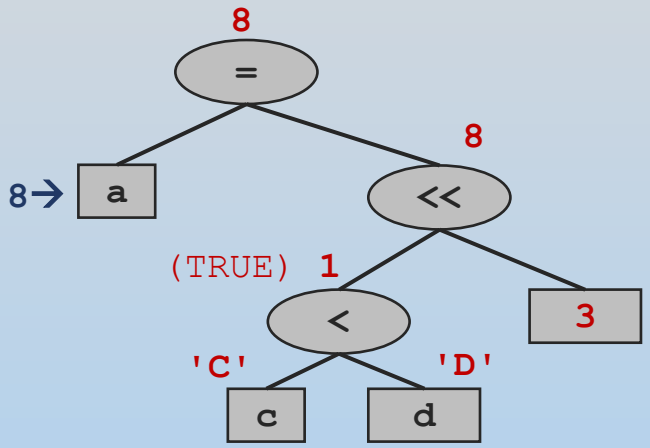
- operators ++_ and --_ return modified value
- operators _++ and _-- return original value

Expression trees

Example 3:

```
char c = 'C', d = 'D';
int  a = (c<d) << 3;
```





Operator precedence

The order in which operators are evaluated:

- Access and Grouping: `()` (function call), `[]`, `.`, `->`
- Unary: `++`, `--`, `!`, `()` (type casting), `&` (address), `*` (dereference)
- Arithmetic: `*` (multiplication), `/`, `%`; and then `+`, `-`
- Bitwise shifts: `<<`, `>>`
- Relational: `>`, `>=`, `<`, `<=`
- Bitwise `&`; and then `^`; and then bitwise `|`
- Logical `&&`; and then logical `||`
- Assignment: `=`, `+=`, `-=`, `*=`, `/=`, etc.

Use of parenthesis is highly encouraged to override precedence and **avoid ambiguity**

Operator precedence – examples

declarations and initializations

```
int    i = 3, j = 3, k = 3;
char   c = 'B';
double x = 0.0, y = 2.3;
```

expression	equivalent expression	value
`i += j && k`	`i += (j && k)`	4
`'A' <= c && c <= 'Z'`	`('A' <= c) && (c <= 'Z')`	1
`x || i && j - 3`	`x || (i && (j - 3))`	0
`i < j && x < y`	`(i < j) && (x < y)` short circuit	0
`i < j || x < y`	`(i < j) || (x < y)`	1

Lecture overview

- Expression trees
- Type conversion ←
- Function definition and invocation
- Function declaration and prototypes

Type Conversions

C freely converts between any two basic types

- Type promotion (widening): "small" → "large" types

  E.g.: `char` → `int` or `int` → `double`

  (no loss of information)

- Type demotion (narrowing): "large" → "small" types

  E.g.: `long` → `int` or `float` → `int` or `int` → `float`

  (possible loss of information)

Unlike in Java, type demotions are allowed even without explicit casting.

No compilation error messages or warnings!

int
double
char

Type Conversions

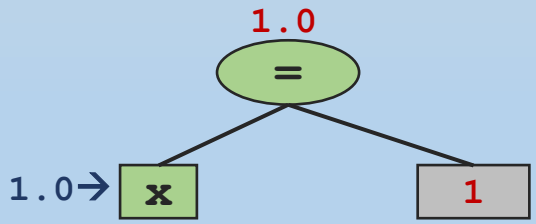
Type conversions of both types will be triggered in any one of the following cases:

- Explicit conversion by casting
- Assignment / function parameter passing
- Mixing types in arithmetic/relational operations

Examples:

```
double x = 1;
char c = 66;
c = 300;
int i = 'A' + 5;
x = 3/4;
x = 3/(double)4;
i = 3 + (int)x;
```

Promotion through assignment



int
double
char

Type Conversions

Type conversions of both types will be triggered in any one of the following cases:

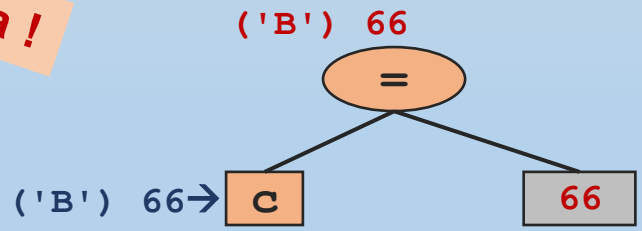
- Explicit conversion by casting
- Assignment / function parameter passing
- Mixing types in arithmetic/relational operations

Examples:

```
double x = 1;
char c = 66;
c = 300;
int i = 'A' + 5;
x = 3/4;
x = 3/(double)4;
i = 3 + (int)x;
```

! Unlike Java !

Demotion through assignment



int
double
char

Type Conversions

Type conversions of both types will be triggered in any one of the following cases:

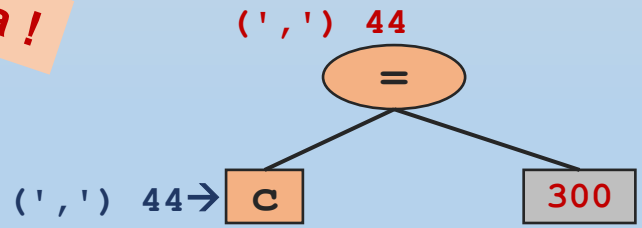
- Explicit conversion by casting
- Assignment / function parameter passing
- Mixing types in arithmetic/relational operations

Examples:

```
double x = 1;
char c = 66;
c = 300;
int i = 'A' + 5;
x = 3/4;
x = 3/(double)4;
i = 3 + (int)x;
```

! Unlike Java !

Demotion through assignment



- Demotion to integer type of small size leads to truncation (truncation means that LSBs are kept)

int
double
char

Type Conversions

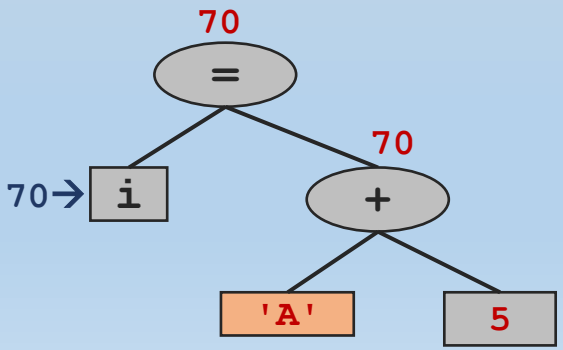
Type conversions of both types will be triggered in any one of the following cases:

- Explicit conversion by casting
- Assignment / function parameter passing
- Mixing types in arithmetic/relational operations

Examples:

```
double x = 1;
char c = 66;
c = 300;
int i = 'A' + 5;
x = 3/4;
x = 3/(double)4;
i = 3 + (int)x;
```

Promotion through operator



int
double
char

Type Conversions

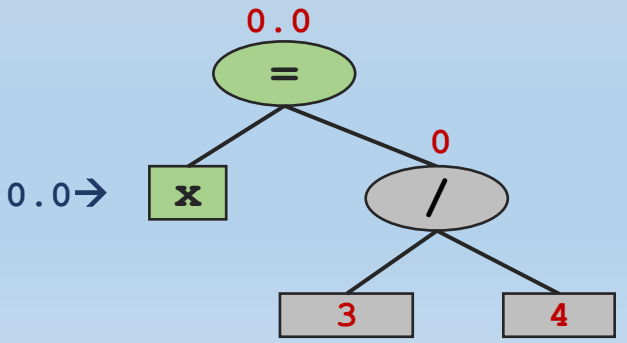
Type conversions of both types will be triggered in any one of the following cases:

- Explicit conversion by casting
- Assignment / function parameter passing
- Mixing types in arithmetic/relational operations

Examples:

```
double x = 1;
char c = 66;
c = 300;
int i = 'A' + 5;
x = 3/4;
x = 3/(double)4;
i = 3 + (int)x;
```

Promotion through assignment



int
double
char

Type Conversions

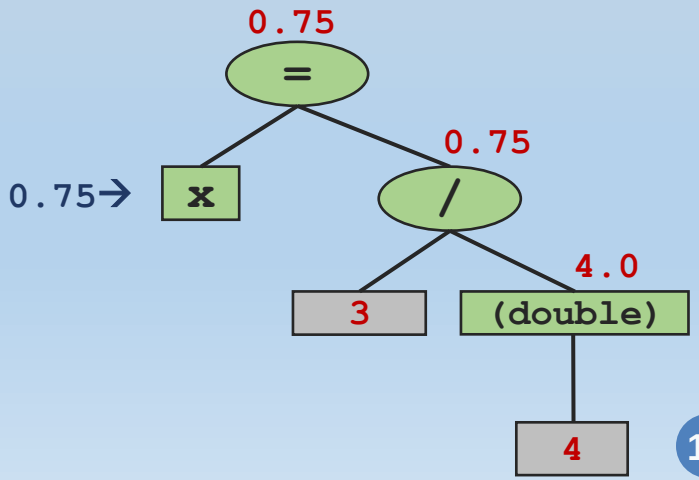
Type conversions of both types will be triggered in any one of the following cases:

- Explicit conversion by casting
- Assignment / function parameter passing
- Mixing types in arithmetic/relational operations

Examples:

```
double x = 1;
char c = 66;
c = 300;
int i = 'A' + 5;
x = 3/4;
x = 3/(double)4;    ←
i = 3 + (int)x;
```

Promotion through casting & operator



int
double
char

Type Conversions

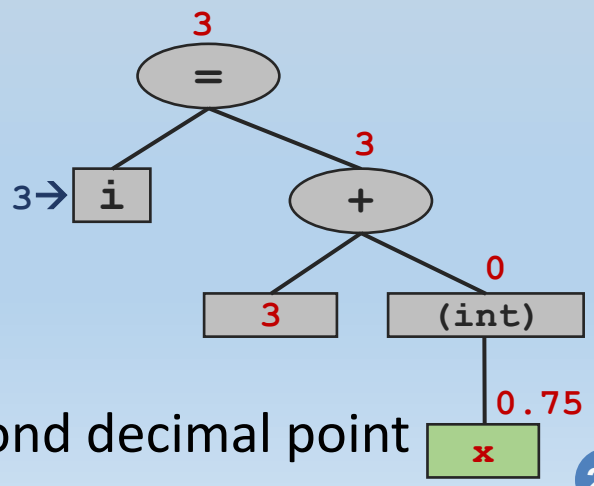
Type conversions of both types will be triggered in any one of the following cases:

- Explicit conversion by casting
- Assignment / function parameter passing
- Mixing types in arithmetic/relational operations

Examples:

```
double x = 1;
char c = 66;
c = 300;
int i = 'A' + 5;
x = 3/4;
x = 3/(double)4;
i = 3 + (int)x;
```

Demotion through casting



- Demotion to integer leads to truncation beyond decimal point

Online poll questions

1. What is the value of 3 | 3 << 3 ?
2. What is the value of 1.0 / 2 + 5 / 2 ?
3. What is the value of (5 && 5/6) + 3 ?
4. Assuming that i=0, what is the value of `i++ && --i`?
5. What is the value of `i` after the expression in the prev. question ?

Operator precedence :

- Access and Grouping: `()` (function call), `[]`, `.`, `->`
- Unary: `++`, `--`, `!`, `()` (type casting), `&` (address), `*` (dereference)
- Arithmetic: `*` (multiplication), `/`, `%`; and then `+`, `-`
- Bitwise shifts: `<<`, `>>`
- Relational: `>`, `>=`, `<`, `<=`
- Bitwise `&`; and then `^`; and then bitwise `|`
- Logical `&&`; and then logical `||`
- Assignment: `=`, `+=`, `-=`, `*=`, `/=`, etc.

Online poll – solutions

What is the value of 3 | 3 << 3 ?

- `1`
- `8`
- `24`
- `27` ←

3 | 3 << 3

= 3 | (3 << 3)

= 3 | $3x2^3$

= 3 | 24 = 27 (no “bits” in common)

Online poll – solutions

What is the value of 1.0 / 2 + 5 / 2 ?

- `2`
- `2.0`
- `2.5` ←
- `3.0`
- `3`

1.0 / 2 + 5 / 2

= (1.0 / 2) + (5 / 2)

double div int div

= 0.5 + 2 = 2.5

Online poll – solutions

What is the value of (5 && 5/6) + 3 ?

- 3 ←
- 4
- 5
- 6
- 8

(5 && 5/6) + 3

= (5 && 0) + 3

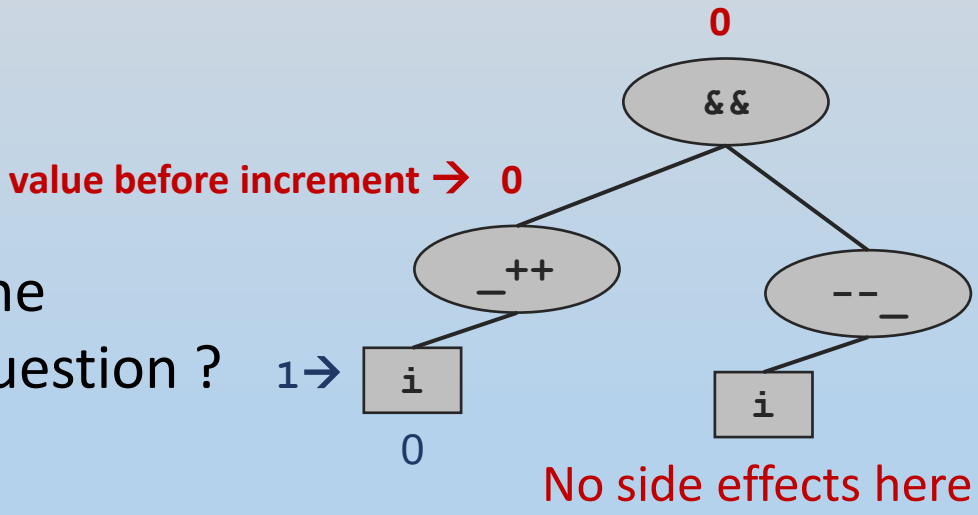
= 0 + 3 = 3

Online poll – solutions

Assuming that i=0, what is the value of `i++ && --i`?

- `0` ←
- `1`
- `2`

Since the first term of the && operator is evaluated to FALSE (0), the second term is not evaluated (short circuit)



What is the value of i after the expression in the previous question ?

- `0`
- `1` ←
- `2`

Type Conversions

Important notes:

1. Types are determined in compilation time (not in runtime)
2. Conversions happen freely between all basic data types
3. Integer demotion to smaller size type truncates the MSBs
4. Floating point demotion to integer truncates decimal LSBs

MSB – most significant bits (left) LSB – least significant bits (right)

Side note:

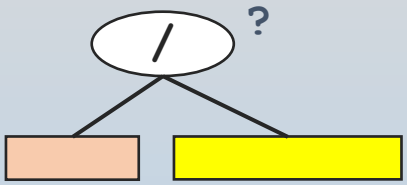
- Type conversion does not happen in **`printf`** and **`scanf`**. Need to use the **specifiers that match the exact type**.

```
printf("%d",3.5);
```
← leads to warning (even without `-Wall`)
and faulty printing (no guarantee of correct runtime output)

Type hierarchy

- Type hierarchy used when converting types in binary operations:
  (from large to small)

```
long double
double
float
unsigned long long
long long
unsigned long
long
unsigned int
int
unsigned short
short
char
```



- ✓ All floating point types are “higher” than all integer types
  (e.g., `float` > `long long`)
- ✓ Every unsigned integer type is “higher” than its signed counterpart
  (e.g., `unsigned long` > `long`)

Type hierarchy

There are some cases where value is modified, or information is lost, even when converting to a type higher in the hierarchy

- Conversion between signed and unsigned types leads to change in value (large unsigned values converted to negative signed values).
- But no information is lost (if you convert back, you get the original number).

```
int a;
unsigned u;

u = a = -1;          // a <- -1 and u <- 4,294,967,295
a = u;               // a <- -1
```

(4,294,967,295 = 2^{32}-1)

Type hierarchy

There are some cases where value is modified, or information is lost, even when converting to a type higher in the hierarchy

- Types **`float`** and **`int`** both use 4 bytes, and conversion between them may lead to loss of info in both directions

```
int a;
float f;

a = f = 3.6;          // f <- 3.6 and a <- 3
f = a;                // f <- 3.0

f = a = 123456789;    // a <- 123456789 and f <- 123456792.0
a = f;                // a <- 123456792
```

`float` has 23 bits of mantissa, so it guarantees relative precision of $2^{-23} \approx 10^{-7}$
so only the top 7 digits of 123,456,789 are guaranteed accurate representation

Lecture overview

- Expression trees
- Type conversion
- Function definition and invocation ←
- Function declaration and prototypes

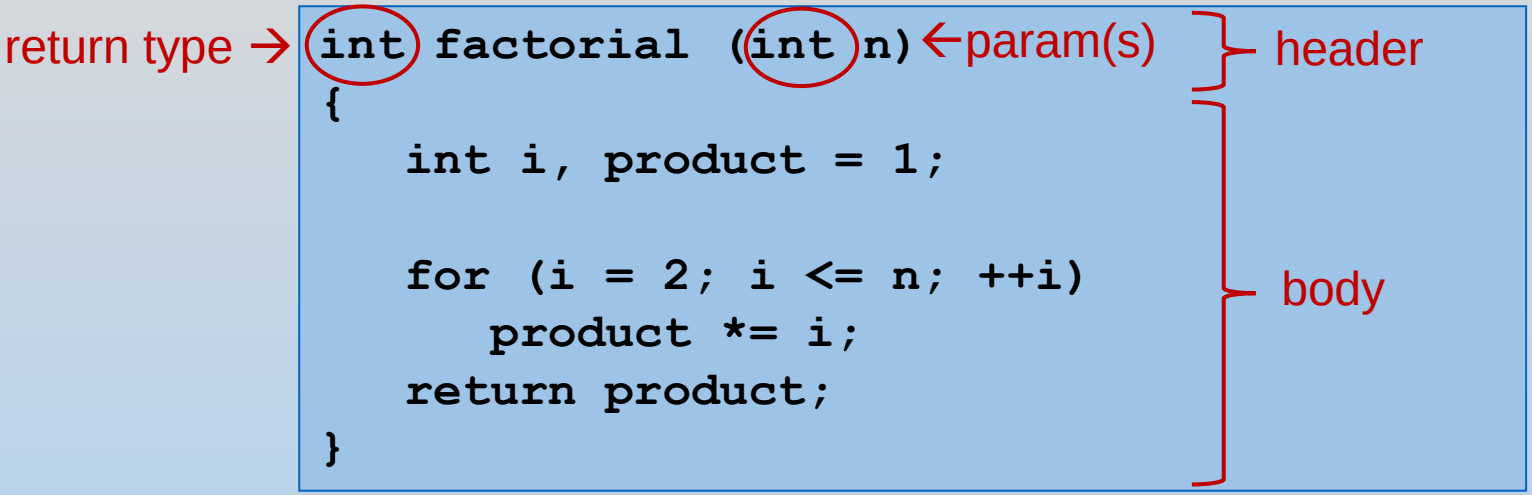
Working with functions

A C program can refer to a function in three different contexts:

- Function declaration
  - function's interface (type of params and return val)
- Function definition
  - function's actual implementation
- Function invocation
  - function call / usage

Function definition

A function definition consists of a header and a body:



The `return` statement

`return` is typically followed by an expression of the **return type**

```
return <expression>;
```

- If a different type is used, **type conversion** is invoked

```
int sum(int a, int b) {
   printf ("%d+%d=%d\n", a, b, a+b);
   return a+b;
}
```

- Standard return form

```
double sum(int a, int b) {
   printf ("%d+%d=%d\n", a, b, a+b);
   return a+b;
}
```

- Return value converted to type `double`

The `return` statement

`return` is typically followed by an expression of the **return type**

```
return <expression>;
```

Implicit actions:

- If the return statement does not include an expression, **the last evaluated expression** is returned
- If a function definition ends without a return statement, **an implicit return** is invoked

The `return` statement

(code in /share/classes/class06/sum.c)

```
int sum(int a, int b) {
   printf ("%d+%d=%d\n", a, b, a+b);
   return a+b;
}
```

- Standard return form

Bad practice !!

```
int sum(int a, int b) {
   printf ("%d+%d=%d\n", a, b, a+b);
   return;
}
```

- Implicit return value – returning last expression (length of printed string)

```
int sum(int a, int b) {
   printf ("%d+%d=%d\n", a, b, a+b);
}
```

- Implicit return statement – returning last expression (length of printed string)

➔ It's good programming practice to always use an explicit return statement of the correct types !!

Functions with no parameters / return value

If you want to define a function that receives no parameters or does not return a value, use **`void`**

```
void welcome (void) {
   printf ("Welcome!\n");
}
```

Avoid the following:

```
welcome () {
   printf ("Welcome!\n");
}
```

(will actually return an int – the length of the printed string)

It's good programming practice to always define your return type and full parameter list **!!**

Functions with no parameters / return value

If you want to define a function that receives no parameters or does not return a value, use **`void`**

```
void welcome (void) {
   printf ("Welcome!\n");
}
```

Do not use implicit function / parameter definitions !

It's good programming practice to always define your return type and full parameter list **!!**

Function invocation

- Functions are invoked by writing the function name and then writing the arguments in parenthesis, e.g., `sum(3,4)`
- Function calls (invocations) act like any other expression evaluation (evaluation, side effects, type conversions)

```
int sum(int a, int b) {
   printf ("%d+%d=%d\n", a, b, a+b);
   return a+b;
}

int main() {
   int a,b;
   scanf("%d%d", &a, &b);
   sum(a,b); //returned value not stored
   return 0;
}
```

- The special `main()` function is invoked automatically when the program starts

Every C program must contain exactly one main function!

Parameter passing – by value

- In C, arguments to functions are always **passed by value**
- When an expression is passed as an argument, it is first evaluated, and only the result is passed to the function
- **The variables passed as arguments are not changed in the calling environment**
- More on the mechanism next lecture …

Parameter passing – by value

```
int compute_sum (int n) {     /* sum from 1 to n */
   int sum = 0;
   for ( ; n > 0; n--)        /* n is changed */
      sum += n;
   return sum;
}
```

Parameter passing – by value

```
int compute_sum (int n) {     /* sum from 1 to n */
   int sum = 0;
   for ( ; n > 0; n--)        /* n is changed */
      sum += n;
   return sum;
}

int main() {
   int n = 3, sum;
   printf("%d\n", n);         /* 3 is printed */
   sum = compute_sum (n);
   printf("%d\n", n);         /* 3 is printed */
   printf("%d\n", sum);       /* 6 is printed */
   return 0;
}
```

- The `main` variable `n` and the `compute_sum` parameter `n` have different storage units
- During invocation, the variable value is assigned to the parameter

Online poll

1. Consider the function **`sum`** with the following definition:

```
int sum(int a, double b) {
  return a + b;
}
```

What is the value of `x` after the following statement:

```
double x = sum(2.5 , 3.5);
```

2. Consider the function **`inc`** with the following definition:

```
int inc(int a) {
  return a++;
}
```

What are the values of `a` & `b` after the following statements:

```
int a = 3,  b = inc(a);
```

Online poll – solutions

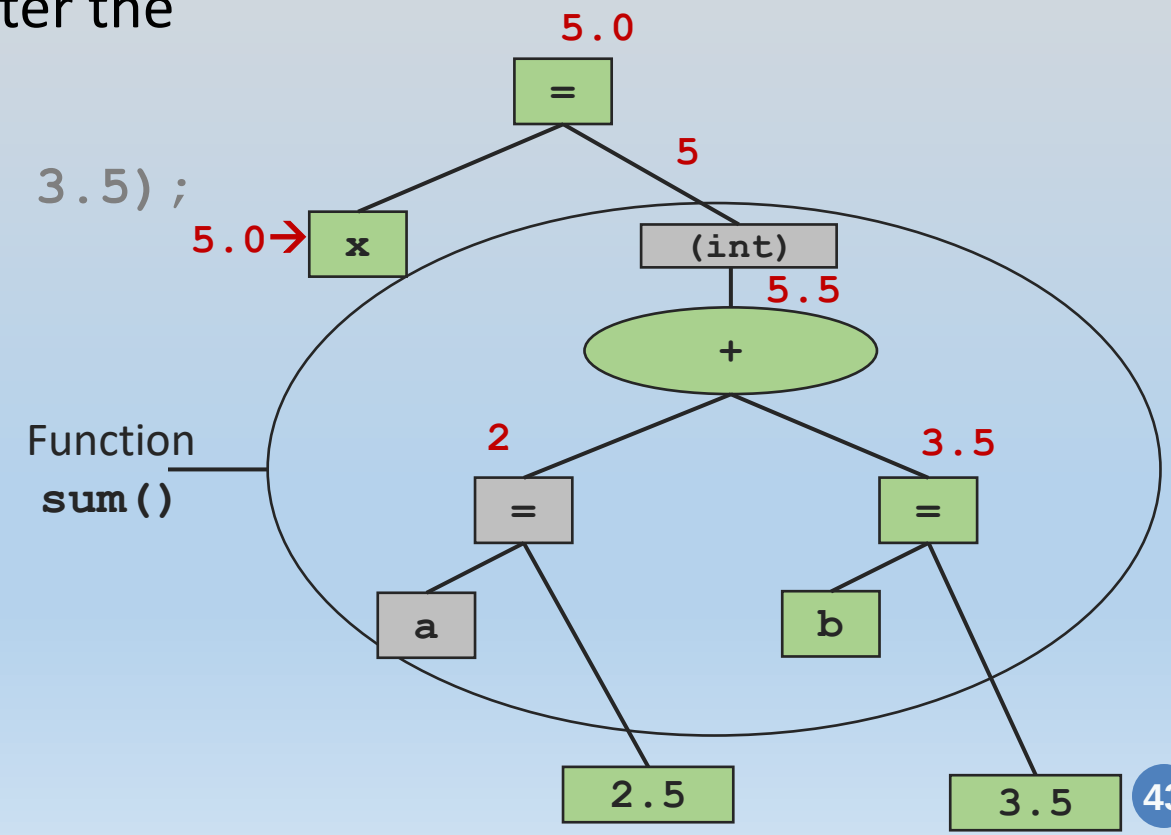
Consider the function **sum** with the following definition:

```
int sum(int a, double b) { return a + b; }
```

What is the value of `x` after the following statement:

```
double x = sum(2.5 , 3.5);
```

- 5
- 5.0 ←
- 5.5
- 6
- 6.0



Online poll – solutions

Consider the function **`inc`** with the following definition:

```
int inc(int a) { return a++; }
```

What are the values of `a` & `b` after the following statements:

```
int a = 3,  b = inc(a);
```

- **Program will not compile**
- **a = 3 b = 3** ←
- **a = 3 b = 4**
- **a = 4 b = 3**
- **a = 4 b = 4**

- Variable a of `inc()` is incremented but this does not affect the values of variables a & b in calling function.

Lecture overview

- Expression trees
- Type conversion
- Function definition and invocation
- Function declaration and prototypes ←

Function declaration

The C compiler requires that every function be **declared before it is invoked**

```
int main() {
   int a,b;
   scanf("%d%d", &a, &b);
   sum(a,b);
   return 0;
}
```

← invocation

Function declaration

The C compiler requires that every function be **declared before it is invoked**

```
int sum(int a, int b) {
   printf ("%d+%d=%d\n", a, b, a+b);
   return a+b;
}

int main() {
   int a,b;
   scanf("%d%d", &a, &b);
   sum(a,b);
   return 0;
}
```

← declaration
← definition
← invocation

- To ensure function declaration is before invocation: option 1 is order functions by their "call graph"

Function declaration

The C compiler requires that every function be **declared before it is invoked**

```
int sum(int, int);                          ← declaration

int main() {
   int a,b;
   scanf("%d%d", &a, &b);
   sum(a,b);                                ← invocation
   return 0;
}

int sum(int a, int b) {                     ← declaration
   printf ("%d+%d=%d\n", a, b, a+b);        ← definition
   return a+b;
}
```

- To ensure function declaration is before invocation: option 2 (the “better” option) is to separate function declarations from their definitions

Function prototype

The function declaration can be separated from its definition by using a **function prototype**

- Prototypes specify the parameter types and return types (like the header of a function definition)
- However, function prototypes do not have to specify parameter names

```
float maximum (float, float);
void  print_info(void);
```

- At the time of function definition, give params names

```
float maximum (float x, float y) {...}
void  print_info() {...}
```

Function prototype

- Prototypes specify the parameter types and return types (like the header of a function definition)
- However, function prototypes do not have to specify parameter names
- At the time of function definition, give params names

Function prototype

- Prototypes specify the parameter types and return types (like the header of a function definition)
- However, function prototypes do not have to specify parameter names
- At the time of function definition, give params names
- But:

! Unlike Java !

No function overloading/overriding!

A function label (name) can be used by no more than one function in your program.

Function declaration vs. definition

- A *function declaration* tells the compiler that a function exists, and what are its parameter types and return type.
- A *function definition* is the actual C code that describes what the function does.
  - Each function should **be defined exactly once**, but it can be **declared several times** (the definition contains an implicit declaration).
  - No function name overriding in C.
  - The definition and (all) declaration(s) should be 100% type consistent.
  - Invocations do not have to be type consistent – type conversions are allowed.

Function prototype - example

```
int sum(int, int);                                  ← declaration


int main() {
   int a,b;
   scanf("%d%d", &a, &b);
   sum(a,b);                                        ← invocation
   return 0;
}

int sum(int a, int b) {                             ← declaration
   printf ("%d+%d=%d\n", a, b, a+b);                ← definition
   return a+b;
}
```

Function prototype - example

```
int sum(int b, int a);
int sum(int, int);
int sum(int c, int);

int main() {
   int a,b;
   scanf("%d%d", &a, &b);
   sum(a,b);
   return 0;
}

int sum(int a, int b) {
   printf ("%d+%d=%d\n", a, b, a+b);
   return a+b;
}
```

← declarations

← invocation

← declaration

← definition

- Multiple declarations are possible if they all agree on data types

Function prototype - example

```
int sum(int b, int a);
int sum(int, int);
int sum(int c, int);

int main() {
   double a,b;
   scanf("%lf%lf", &a, &b);
   sum(a,b);
   return 0;
}

int sum(int a, int b) {
   printf ("%d+%d=%d\n", a, b, a+b);
   return a+b;
}
```

← declarations

← invocation

← declaration

← definition

- Multiple declarations are possible if they all agree on data types
- Type conversion is fine during copying of parameters

Implicit function declaration

If the compiler encounters a function invocation with no previous declaration of that function, it assumes that this function returns type `int`

Bad practice !!

```
#include<stdio.h>

int main() {
    int i = f(2,3);
    printf("%d\n",i);
    return 0;
}

int f(int a, int b) {
    return a+b;
}
```

- Code compiles fine
  - "implicit declaration" warning (with **`-Wall`**)

Implicit function declaration

If the compiler encounters a function invocation with no previous declaration of that function, it assumes that this function returns type `int`

Will not compile !!

```
#include<stdio.h>

int main() {
    int i = f(2,3);
    printf("%d\n",i);
    return 0;
}

double f(int a, int b) {
    return a+b;
}
```

- Code does not compile
  - error: conflicting types for 'f'
  - "implicit declaration" warning (with **`-Wall`**)

At home…

- 4th homework assignment published today, and is due on June 2
- We also posted a guide for setting up a Linux virtual machine (VM) on your computer, for those of you who wish to work locally and not just on the course server