Notes 5.0: Variables, assignments, operators

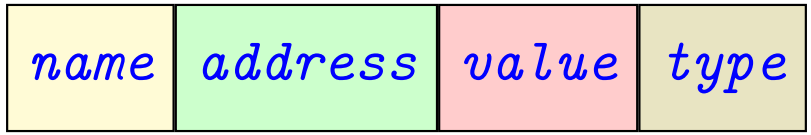
COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2013 session 1

Identifiers

Identifiers are names for data items, that we mentally represent as:



An identifier is a sequence of (lowercase and uppercase) letters, digits and underscores, that does not start with a digit. For instance:

- `cat`, `cat007`, `BigCat`, `Big_cat` and `_cat` are valid identifiers.
- `$cat`, `007cat`, `Big-cat`, `Big cat` and `Big'cat` are invalid identifiers.

Identifiers are case sensitive: an uppercase letter is considered distinct from the corresponding lowercase letter.

The maximal number of characters in an identifier is system dependent. C99 demands at least 63 characters.

Variables and declarations

Identifiers are used to, in particular, give names to data items called variables.

A program creates that data item when the variable is declared, in one of two ways:

- globally, *i.e.*, not in the body of any function, in which case the variable is said to be global;
- locally, *i.e.*, in the body of some function, in which case the variable is said to be local.

A declaration is a particular kind of statement that like any other statement, has to end in a semicolon. The rest of the statement consists of mention of the type of the variable, followed by the variable name.

Declaring a global variable automatically sets its value to a default value, as can be verified by running declare.c.

The address of a variable (1)

The address of variable can be accessed, and is often useful to access. The & unary operator can take a variable name as argument, and return the address of that variable, that can be displayed by `printf()` using the `%p` (where `p` stands for pointer) format specifier.

For instance, a program whose `main()` function is

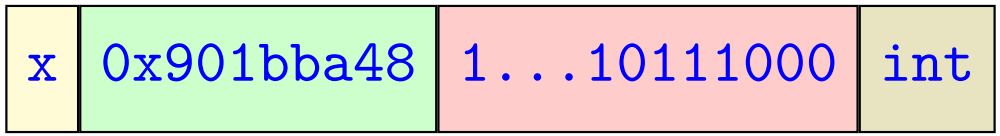
```
int main(void) {
    int x;
    printf("Variable x has address %p.\n"
           "Its current value is %d\n", &x, x);
    return EXIT_SUCCESS;
```

might yield as an output:

```
Variable x has address 0x901bba48.
Its current value is -1073744968
```

The address of a variable (2)

From the previous output, we can depict that variable as follows.

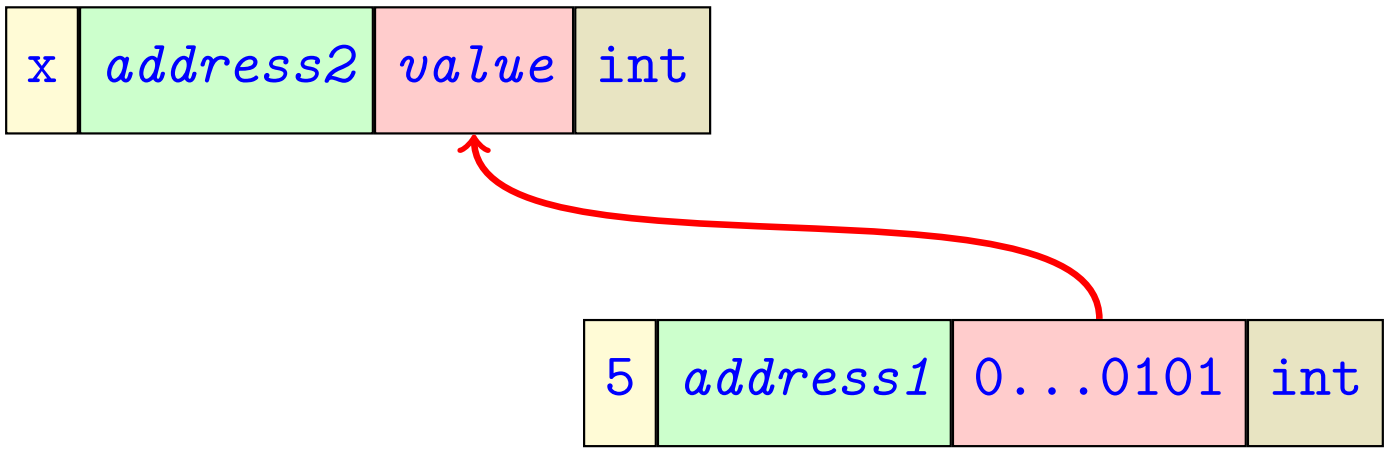


The value of a variable can change over time during program execution; its address cannot change.

Assignments (1)

A variable can be assigned the value of a data item ι, thanks to an assignment statement, consisting of that variable name, the assignment operator, `=`, that data item's name, and the final semicolon.

The name of ι can be a constant, such as in the assignment `x = 5;`



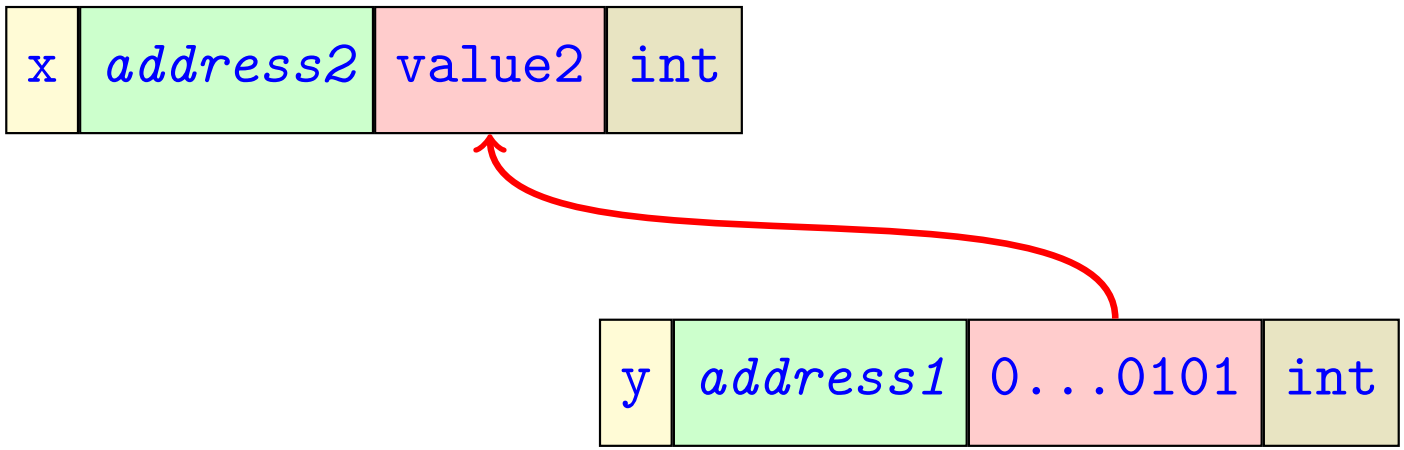
An assignment is actually a side effect of the evaluation of an expression; the evaluation of `x = 5;` is a data item of type `int`, whose value is (the internal representation of the integer) 5.

Assignments (2)

The name of ι can be another variable, such as a variable of type `int`, in an assignment such as

```
x = y;
```

Assuming that the current value of (the data item whose name is) `y` is (the internal representation of) 5, this assignment can be depicted as:



Declarations and assignments combined

It is possible to declare many variables of the same type in the same statement, and also assign some values to some or all of these variables, with a comma before each new variable except the first one, as in

```
int x = 0, y, z = 0;
```

where `x`, `y` and `z` are declared, `x` and `z` are initialised to `0`, and `y` is not initialised, as a side effect of the evaluation of an expression that results in a data item of type `int` and of value 0.

What does assigning to a variable of type *type1* the value of a variable of type *type2* mean if *type1* and *type2* are different?

The question is pertinent even if *type1* and *type2* allocate the same number of bits to represent a value, as they might use a different coding scheme.

Assignments between signed or unsigned types

Let variable v and data item ι be of signed or unsigned type, with the value of ι assigned to v.

- If it is possible to represent the value of ι in the type of v, then a conversion that respects the mathematical values is performed.
- If v is unsigned and ι's value cannot be represented in the type of v, then the value assigned to v is obtained from ι's value as follows.
  - If at least as many bits are allocated to the type of v as to the type of ι, then the value of ι is negative; add as many high order 1s as needed to the left of the value of ι so that the resulting number of bits equals the number of bits allocated to the value of v (which would preserve the value of ι if v was actually signed);
  - If fewer bits are allocated to the type of v than to the type of ι, then get rid of the high order bits that do not fit.
- In all other cases, something happens that we happily ignore.

See integer_to_integer_assignments.c.

Assignments between floating point types

Let variable v and data item ι be of type `float`, `double` or `long double`, with the value of ι assigned to v.

- If the type of ι is smaller than the type of v, then the mathematical value of ι is preserved.
- If the type of ι is greater than the type of v but the value of ι is within the range of values that characterises the type of v, then the result is one of the two mathematical values that is closest to the value of ι and can be represented in the type of v.
- Otherwise, something happens that we happily ignore.

See real_to_real_assignments.c.

Other assignments

- Let variable v be of a type meant to represent (signed) integers, and let data item ι be of type `float`, `double` or `long double`, with the value of ι assigned to v. If the integral part of the mathematical value of ι can be converted to a value of type the type of v, then the conversion gives the expected result.
- Let variable v be of type `float`, `double` or `long double`, and let data item ι be of a type meant to represent (signed) integers. If the value of ι is within the range of values that characterises the type of v, then the result is one of the two mathematical values that is closest to the value of ι and can be represented in the type of v.
- A variable of type `bool` receives a value of `0` when assigned a value of any type that has all bits set to `1`, and a value of `1` when assigned a value of any type that has at least one bit set to `0` (intuitively, zero is false and anything not equal to zero is true).

constant variables

The declaration of a variable can be prefixed with the keyword `const`, making the variable read-only: its value cannot be modified after it has been declared.

A program containing the following code fragment does not compile.

```
const int n = 17;
n = 18;
```

- `const int n;` would not generate any warning, even with the `-Wall` option to the compiler, though it is likely to be the source of a bug if that declaration is local and `n` is involved in a computation.
- `#define COURSE_CODE 9021` and `const int 9021;` achieve the same effect; it is common practice to use all upper case letters for macro names, and only for macro names, but this is just a convention.
- It is good practice to give variables least privilege and declare all variables whose value is not meant to be modified as `const`.

Arithmetic operators (1)

There are 5 arithmetic operators:

- The multiplicative operators: `*` for multiplication, `/` for division, and `%` for remainder.
- The additive operators: `+` for addition and `-` for subtraction.

The multiplicative operators take precedence over the additive operators, and associate left-to-right; to change the order of evaluation given by these priority and associativity rules, parentheses are used.

- `2 * 3 - 4` evaluates to `2`
- `2 * (3 - 4)` evaluates to `-2`
- `6.0 / 4.0 / 2.0` evaluates to `0.75`
- `6.0 / (4.0 / 2.0)` evaluates to `3.0`

These operators operate on data items of the same type, and `%` does not operate on data items of type `float`, `double` or `long double`.

Arithmetic operators (2)

The type of the result of an arithmetic operation is equal to the type of the operands; in particular, integer division yields an integer:

- `6.0 / 4.0` evaluates to `1.5`.
- `6 / 4` evaluates to `1`.

More facts about division should be noted.

- When two integers are divided, the decimal part, if any, is discarded (truncation toward zero).
- When two integers m and n are divided, the mathematical equality $(m/n) * n + m\%n = m$ is meant to hold.
- Integer division by `0` or `0.` might yield `0`, or `inf`, or a run-time error, depending on the compiler; it should not be attempted.

Operations between data items of an unsigned type that allocates n bits to the representation is congruent modulo 2^n to the result of the operation: use show_bits.c to see why `4000000000U * 2U` evaluates to `3705032704`.

Combined division and remainder calculation

The `stdlib()` header file defines the `div_t`, `ldiv_t` and `lldiv_t` types—structures with two fields, `quot` and `rem`, of type `int`, `long` and `long long`, respectively—, and the functions

```
div_t div(int, int);
ldiv_t ldiv(long, long);
lldiv_t lldiv(long long, long long);
```

that given arguments `n` and `d`, compute `n / d` and `n % d` more efficiently than by evaluating the latter two expressions.

For instance, given the statement

```
div_t x = div(17, 3);
```

`x.quot` is an `int` that evaluates to `5` and `x.rem` is an `int` that evaluates to `2`.

Automatic conversion of operands (1)

When a binary operator, such as all arithmetic operators, is applied to operands that are not all of the same type, automatic conversion of the operands to data items of the same type is performed before the operation is applied, according to the following two-step transformation.

The first step can be described as follows.

- Operands of type `signed short` are converted to `signed int`.
- Operands of type `unsigned short` are converted to `int`s if all values of type `unsigned short` can fit into an `int` (because more bytes are allocated to `int`s than to `unsigned short`s), and to `unsigned int`s otherwise.
- Depending on the implementation, `float`s might or might not be converted to `double`s.

Automatic conversion of operands (2)

The second step can be described as follows.

- If at least one operand is of type `long double`, `double` or `float` then both operands are converted to the largest of `long double`, `double` or `float` that is the type of some operand. For instance,
  - two operands of respective types `long long int` and `float` are converted to two operands of type `float` or `double`, depending on the outcome of the first step;
  - two operands of respective types `float` and `long double` are converted to two operands of type `long double`.
- If both operands are of type `unsigned ...`, then both operands are converted to the largest of `unsigned long long`, `unsigned long` or `unsigned` that is the type of some operand.
  For instance, two operands of respective types `unsigned int` and `unsigned long` are converted to two operands of type `unsigned long`.

Automatic conversion of operands (3)

- If both operands are of type `signed ...`, then both operands are converted to the largest of `signed long long`, `signed long` or `signed` that is the type of some operand.
  For instance,
  - two operands of type `signed short` are converted to two operands of type `signed int`;
  - two operands of respective types `signed int` and `signed long` are converted to two operands of type `signed long`.
- If one operand is of type `unsigned ...`, the other of type `signed ...`, and the signed type can represent all values of the unsigned type then both operands are converted to the signed type.
  For instance, if more bits are allocated to a `long long` than to an `int`, then two operands of respective types `unsigned int` and `signed long long` are converted to two operands of type `signed long long`.

Automatic conversion of operands (4)

If one operand is of type `unsigned ...`, the other of type `signed ...`, and the signed type cannot represent all values of the unsigned type, then the internal representations do not change and the types of both operands become either the unsigned type or the unsigned version of the signed type, whichever is of higher rank.

For instance, if the same number of bits are allocated to `int`s and to `long`s, but not to `long long`s, then

- two operands of respective types `unsigned int` and `signed long` become two operands of type `unsigned long`, with the internal representations being left unchanged;
- two operands of respective types `unsigned long long` and `signed long` become two operands of type `unsigned long long`, with the internal representations being left unchanged.

The program conversions.c illustrates some of the conversion rules.

Round off errors

When performing computations on floating point numbers, the conversion rules do not tell the whole story. When one knows the internal representation of floating point numbers, it comes as no surprise that on some machines, `b is 0.000000e+00` is the output of

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
int main(void) {
    float a = 2.0e20 + 1.0;
    float b = a - 2.0e20f;
    printf("b is %e\n", b);
    return EXIT_SUCCESS;
}
```

Operations on characters

Arithmetic operators operate on data item of type `char` as they would on data items of a type meant to represent integers. For instance, `a d H` is the output of the following program, where the `%c` format specifier is used by `printf()` to display characters.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    char x = 'a';
    printf("%c", x);
    x = x + 3;
    printf(" %c", x);
    x = x + 'A' - 'a' + 4;
    printf(" %c\n", x);
    return EXIT_SUCCESS;
}
```

The cast operator

An expression of the form *(type)name_of_data_item* creates a new data item of type *type*, whose value is obtained from the value of the data item whose name is *name_of_data_item* by applying the conversion rules that govern the behaviour of the assignment operator. We describe that effect of expression as casting *name_of_data_item* to *type*.

For instance:

- `(int)29.3 + (int)11.7` evaluates to 40
- `(int)(29.3 + 11.7)` evaluates to 41
- `(int)29.3 + 11.7` evaluates to 40.7
- `(float)6 / (float)4` evaluates to 1.5
- `(float)(6 / 4)` evaluates to 1.0
- `(float)6 / 4` evaluates to 1.5

Increment and decrement operators (1)

The unary increment and decrement operators increase and decrease the value of their operand by one, respectively.

These operators come in two versions: prefix and postfix

When used in simple statements, both versions are similar: `a = 1, b = 1` is the output of

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int a = 0, b = 0;
    a++;
    ++b;
    printf("a = %d, b = %d \n", a, b);
    return EXIT_SUCCESS;
}
```

Increment and decrement operators (2)

When one of these operators is part of a larger expression,

- the prefix version does its job *before* the expression is evaluated;
- the suffix version does its job *after* the expression is evaluated.

This is illustrated in ___crement.c

- `(x * y)++` is *invalid*: these operators can only be applied to variables.
- These operators should not be applied to a variable that appears more than once in an expression: dealing with `p = n * (n++ / 2)`, some compilers would first evaluate `n`, others would first evaluate `n++ / 2`, so the actual result cannot be known without knowing how the compiler works, which is not meant to be known as programmers should strive to write portable programs.

Relational operators

There are 6 relational operators: `==` for equality (not to be confused with the assignment operator, `=`), and a number of operators for inequality, namely `!=`, `<=`, `<`, `>` and `>=`, to test whether two values are distinct, or if one value is smaller than or equal to, strictly smaller than, strictly greater than, or greater than or equal to the other.

Relational operators returns `true` or `false`.

All these operators associate left-to-right, they all have lower precedence than the arithmetic operators, higher precedence than the assignment operators, with `==` and `!=` of lowest, identical priority, and with the other four relational operators of higher, identical priority.

For (an awful) example, `true` would be assigned to both `a` and `b` with

```
bool a = 0 == 1 == 0;
bool b = 0 == 3 < 0;
```

Boolean operators

There are 3 boolean operators: the unary `!` for negation, and the binary `||` and `&&` for inclusive disjunction and conjunction, respectively.

Boolean operators returns `true` or `false`.

All these operators associate left-to-right, with

- `!` having higher precedence than any binary operator,
- `&&` having lower precedence than the relational operators, and higher precedence than `||`, and
- `||` having higher precedence than the assignment operators.

Evaluation stops as soon as possible. Hence both expressions below would evaluate to `true`, with no division by zero being performed.

```
!(2 == 3 && 1 / 0 == 3);
2 == 2 || 1 / 0 == 3;
```

Bitwise operators

There are 3 bitwise operators: the unary ~ for bitwise negation, and the binary |, &, ^, << and >> for bitwise disjunction, bitwise conjunction, bitwise xor, left shift and right shift, respectively.

All these operators associate left-to-right, with

- ! having higher precedence than any binary operator,
- & having lower precedence than the relational operators, and higher precedence than ^,
- ^ having higher precedence than |, and
- | having higher precedence than the boolean operators.

The program bits.c illustrates.

More on assignment operators

Also referred to as assignment operators, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=` and `>>=` are abbreviations for assigning to the variable on the left hand side of the operator the result of applying that operator to the current value of that variable and to the name of a data item provided on the right hand side of the `=` sign.

For instance, `x *= 3 * y + 12` is an abbreviation for

```
x = x * (3 * y + 12)
```

The abbreviated forms are more compact and may produce more efficient machine code than their longer counterparts.

Assignment operators associate right-to-left. For (a bad) example,

```
int a = 2, b;
a += b = 1;
```

would assign `3` to `a` (a side effect of the 2nd expression being evaluated to 3).

Macros for some operators

The `iso646.h` header file defines the following macros for alternative denotations of some inequality operator, the boolean operator, and some bitwise and assignment operators.

- `not_eq` for `!=`
- `not` for `!`
- `or` for `||`
- `and` for `&&`
- `compl` for `~`
- `bitor` for `|`
- `bitand` for `&`
- `bitxor` for `^`
- `or_eq` for `|=`
- `and_eq` for `&=`
- `xor_eq` for `^=`