C++ Aide Memoir

Ross Bannister, March 2004/April 2004/ May 2011

Reserved Words

```
asm, auto, break, case, char,
  class, const, continue, default,
  delete, do, double, else, enum,
  extern, float, for, friend, goto,
  if, inline, int, interrupt, long,
  main, naked, new, operator,
  private, protected, public,
  register, return, short, signed,
  sizeof, static, struct, switch,
  template, this, typedef, union,
  unsigned, virtual, void,
  volatile, while.
```

Main Body of Code

Function declarations (not definitions)

Global variable declarations

Type definitions

Enumerator definitions (named integer constants)

Structure definitions

Union definitions

```
int main()
{ statements; ...return integer; }
```

integer=0 (normal ending)

Data Types

Primitive data types

`char` (1 byte integer for characters)

`int` (2-byte signed integer)

`float` (4-byte real)

`double` (8-byte real)

- Logical types are represented as integers (0=false, other value=true)
- Variables can be initialized in the declaration.

Type modifiers

`const` (definition of constants)

`long` (double length integer)

`unsigned` (unsigned integer)

`static` (in functions, maintains value between calls)

Casting (type conversion)

`int`(floatvariable)

Forcing numerical values

`9.99f` (forcing to type float)

`999L` (forcing to long (4-byte) integer)

Derived Data Types

Type definitions

```
typedef variable_type NAME;
```

- variable_type contains a primitive type with type modifiers.
- NAME becomes a pseudonym for variable_type for variable declaration:

```
NAME variable_name;
```

Enumerators

```
enum name
{ integer_identifier1 [= value1],
  integer_identifier2 [= value2],
  ... }
[enum_variable_names];
```

- In the program, a variable (declared as type enum_variable_name) can be set to one of the identifiers. If values are omitted then they take on incremental integers, starting with 0. Further declarations can be made later using name as the type.

Structures

- Structures are user defined composite types, defined by:

```
struct struct_name
{ member_declarations; }
[variable_names];
```

- Member_declarations are variable declarations (primitive, arrays, derived, pointers, etc.).
- Other variables of this type can be declared:

```
struct struct_name variable_name;
```

- It is possible to have arrays of derived data types.
- Members of the structure are referenced by:

```
variable_name.member_name
```

Union

```
union name
{ member_declarations; }
[struct_variable_names];
```

- As `struct`, but all members share the same memory.

Comments

`/* ... */` (multi-line comments)

`// ...` (single-line comments)

Operators

Arithmetic operators

`+, -, *, /` (ordinary arithmetic operators)

`%` (remainder)

`++, --` (increment, decrement)

Logical operators

`&&` (and), `||` (or), `^` (xor), `!` (not)

Relational operators

`==, !=, <, >, <=, >=`

Compound assignment operators

```
expression1 operator =
  expression2;
```

is equivalent to

```
expression1 = expression1 operator
  expression2;
```

Control structures

If ... else

```
if (expression) {statement1;} else
  {statement2;}
```

The else branch is optional

This is equivalent to

```
expression1 ? statement1 :
  statement2
```

Switch

```
switch (integer_expression)
{ case integer1:
    statements1;
    break;
  case integer2:
```

```
    statements2;
    break;
  default:
    statements3;
}
```

While

```
while (logical_expression)
{ statements; }
```

do ... while

do

```
{ statements; }
while (logical_expression);
```

For

```
for (statement1; expression;
  statement2)
{ statements; }
```

is equivalent to

```
statement1;
while (expression)
{ statements;
  statement2; }
```

Common loop usage

```
for (i=1; i<=100; i++)
{ statements; }
```

- If statement1 or statement2 is missing, nothing is executed in their place.
- If expression is missing it is true by default.
- Statement1 can include a declaration of the loop variable (it continues to exist after the loop has finished).

Exiting and skipping loops

`break` (exit the loop)

`continue` (skip to next cycle (statement2 is still evaluated in `for` loop)

- The above can be used in `while`, `do ... while` and `for` loops.

Arrays

Declaration

```
type array_name [size];
type array_name [sizex][sizey];
type array_name [size] = {value0,
  value1, ... };
type array_name [sizex][sizey] =
  {{value00, value01, ... },
   {value10, value11, ... },
   { ... } };
```

- Array elements start at 0 and end at `size-1`.
- `size`, `sizex`, `sizey`, etc. can be omitted when declaring and initializing are combined (values are implied).
- Arrays are implicitly pointers to the start of the array.

Using and accessing

```
array_name [index]
array_name [indexx][indexy]
```

Characters and Strings

String declaration and setting

Declaration (with optional initialization)

```
char string_name [] = {char1,
  char2, ..., \0};
char string_name [] = "literal
  string";
```

- Strings are arrays of characters.
- Characters, e.g. `char1`, go inside single quotes.
- Characters inside single quotes evaluate to their ASCII code.
- In the first example declaration above, the last character must be `\0` (see below). This is implied in the second example declaration.
- Literal strings go inside double quotes.
- Literal string arguments can be given in double quotes. This sets up an array and passes the pointer to the start of the array.
- In the above the array length is implied.
- After declaration, strings cannot be set directly using literal strings (use `strcpy` of standard library `string.h`).

```
strcpy (string_name, "literal
  string");
```

Special characters

`\"` double quotes
`\'` single quote
`\\` backslash
`\n` newline
`\0` null (padding)

Input and Output

Keyboard and screen (cout and cin)

```
#include <iostream.h>
...
cout << exp1 << exp2 ...;
cin >> var1 >> var2 ...;
```

Keyboard and screen (formatted)

Output to screen:

```
#include <stdio.h>
...
printf (format_string, exp1, exp2,
  ...);
```

Input from keyboard:

```
scanf (format_string, &exp1, &exp2
  , ...);
```

- The variable arguments for `scanf` are addresses (`&`). The `&` is implicit for pointers or arrays.
- See below for the `format_string` description.

Format string description

Example used with `printf`:

```
printf ("\nThe result for %i is
  %f", int_var, double_var);
```

- The `%i` and `%f` are examples of format specifiers.
- Format specifiers have a general form, `%[flags][width][.precision] [size]type`.
- Only the type part has been used in the example.
- Useful types are given below.

`%c`	character
`%s`	string
`%d` or `%i`	signed integer
`%u`	unsigned integer
`%f`	floating point
`%e` or `%E`	exponential format
`%g` or `%G`	computer chooses `%f` / `%e`
`%x`	hex (lower case letters a-f)
`%X`	hex (upper case letters A-F)
`%p`	pointer

File IO (formatted)

Open a file:

```
#include <stdio.h>
...
FILE* file_var = NULL;
...
file_var = fopen
  (file_name_string, mode_string);
```

- `file_var` is a pointer to a `FILE` type.

- `mode_string` is specified below.
- `file_var` remains `NULL` (logically false) if opening is unsuccessful.

`"r"` reading (file exists)

`"w"` writing (file may or may not exist)

`"r+"` reading and writing (file exists)

`"w+"` reading and writing (file may or may not exist)

Output to file:

```
fprintf (file_var, format_string,
  exp1, exp2, ...);
```

Input from file:

```
fscanf (file_var, format_string, &
  exp1, &exp2 , ...);
```

- The use of `fprintf` and `fscanf` is just like `printf` and `scanf` (above), but with the extra `file_var` parameter.
- See above for the `format_string` description.

Closing a file:

```
fclose (file_var);
```

Functions

Declaration

```
type function_name (arg_type1
  arg_name1, arg_type2 arg_name2,
  ...);
```

- The declaration appears in the header of each file that uses the function.
- The function type and the argument types are primitive data types (with modifiers if necessary), typedefs, enumerators, structures or unions, etc.
- The argument names may be omitted.
- For no arguments, use empty parenthesis. If the function has no type then declare as type `void` (the function is then called as a statement).

Definition

```
type function_name (arg_type1
  arg_name1, arg_type2 arg_name2,
  ...)
{  local variable declarations;
   statements;
   return value;
}
```

- The definition appears after the 'main' function definition.
- The function ends when a `return` statement is found (this can appear many times anywhere in the definition) or at the end.
- No `return` is required if the function is of type `void`.
- Global variables are used by inserting `::` immediately before the variable name (needed only if local variable exists with the same name).
- Parameters are call-by-value.
- Call-by-address is achieved by passing pointers as parameters (see below). This allows any changes of formal parameters (inside the function) to affect the actual parameters (outside).
- All arrays are implicitly pointers to the start of the array.
- Arrays may be passed by using the bare `array_name` as the actual parameters and `array_name[]` in the list of formal parameters.
- For arrays of more than 1-D, the dimensions should be specified.

Pointers

Pointer declaration and null setting

Declaration (with optional setting to the null pointer)

```
type* pointer_name = NULL;
type *pointer_name = NULL;
```

Pointers are always 4 bytes long.

Pointer creation and deletion

```
pointer_name = new type;
delete pointer_name;
```

To delete an entire array

```
delete[] array_name;
```

Accessing information

To access a pointer's target

```
*pointer_name
```

If the pointer is to a structure

```
(*pointer_name).variable_name
```

To return a pointer to a variable (useful for call-by-address)

```
&variable_name
```

Allocation check

```
(pointer_name)
```

will return true if it is an allocated pointer