CS 246 Fall 2019 — Tutorial 5

October 2, 2019

Summary

1 References **1**

2 Dynamic Memory **2**

3 Preprocessor **2**

4 Make and Makefiles **3**

1 References

Syntax:

```
int x = 42;
int &rx = x;
```

A reference acts like an automatically-dereferenced constant pointer to data.

Consider the code below:

```
int x = 10, y = 5;

int &rx = x;
int &ry = y;

int *px = &x;
int *py = &y;

int res1 = (*px + *py) * (*px - *py);
int res2 = (rx + ry) * (rx - ry);
// res1 and res2 contain the same value
```

Note: References cannot refer to nothing — they must always be initialized, and you can't directly initialize them to `nullptr`.

1.1 Pass-by-Reference

Pass-by-value: Makes a copy of the parameter passed for use during the function. Changes to the parameter do not exist outside of the scope of the function.

Pass-by-reference: creates an alias to the parameter. Hence, no copy is made, and changes persist outside the scope of the function. Will only bind lvalues.

Exercise: Which of these lines will compile?

```
int foo(int &x, const int &y) { ... }
int main() {
    int a = 42;
    foo(a, a);   // (A)
    foo(a, 43);  // (B)
    foo(43, a);  // (C)
    foo(43, 43); // (D)
}
```

2 Dynamic Memory

In C++, use `new` instead of `malloc`, and `delete` instead of `free`. To allocate and delete an object on the heap:

```
int *x = new int{5};
...
delete x;
```

To allocate and delete an array on the heap:

```
int *arr = new int[10];
...
delete[] arr;
```

Note: Always use `new` and `delete` together, and use `new[]` and `delete[]` together.

3 Preprocessor

The preprocessor runs before the compiler. It handles all preprocessor directives (lines beginning with `#`).

Common preprocessor directives:

`#include "file.h"`	inserts the contents of `file.h`
`#define var val`	defines a preprocessor macro `var` with value `val`. If `val` is omitted the value is the empty string
`#ifdef var`	includes text until matching `#endif` if `var` is defined.
`#ifndef var`	similar to `#ifdef`, but includes code if `var` is not defined

3.1 Include Guards

At the top of a header file:

```
#ifndef __SOME_UNIQUE_HEADER_NAME
#define __SOME_UNIQUE_HEADER_NAME

<the header code>

#endif
```

4 Make and Makefiles

When compiling projects consisting of more than one file, the `make` utility should be used. `make` requires a *Makefile* to run, consisting of a list of rules detailing how a project should be built.

Basic Makefile:

```
vec3d: main-vec3d.o vec3d.o
    g++ -std=c++14 main-vec3d.o vec3d.o -o vec3d

main-vec3d.o: main-vec3d.cc vec3d.h
    g++ -std=c++14 -c main-vec3d.cc

vec3d.o: vec3d.cc vec3d.h
    g++ -std=c++14 -c vec3d.cc
```

Note: The whitespaces before the build command (in this case, `g++ ...`) **MUST** be a tab.

Common practice: add a phony target `clean` to remove all generated files. Then `make clean` will delete all the generated files.

Second Makefile example:

```
CXX=g++                                 # compiler command
CXXFLAGS=-std=c++14 -Wall -Werror -g # compiler options to pass
EXEC=vec3d                              # name of executable
OBJECTS=main-vec3d.o vec3d.o            # object files

# This one doesn't have a generic form, so have to list it
${EXEC}: ${OBJECTS}
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

# Uses the recipe ${CXX} ${CXXFLAGS} -c main-vec3d.cc -o main-vec3d.o
main-vec3d.o: main-vec3d.cc vec3d.h
```

```
vec3d.o: vec3d.cc vec3d.h

clean:
    rm ${OBJECTS} ${EXEC}

.PHONY: clean
```

The compiler can also generate the depencies for us so that we don't need to update the makefile every time we change them; run `g++` with the `-MMD` option.

Final Makefile:

```
CXX=g++
CXXFLAGS=-std=c++14 -Wall -Werror -g -MMD
EXEC=vec3d
OBJECTS=main-vec3d.o vec3d.o
DEPENDS=${OBJECTS:.o=.d}

${EXEC}: ${OBJECTS}
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

-include ${DEPENDS}

clean:
    rm ${OBJECTS} ${EXEC} ${DEPENDS}

.PHONY: clean
```