An Introduction to C
2501ICT/7421ICT

René Hexel
School of Information and Communication Technology
Griffith University

Semester 1, 2012
Outline

1. Overview
   - Motivation for C, C++, and Objective-C
   - The C Programming Language

2. Compiling and Makefiles
   - Using the Command Line compiler
   - Creating and using Makefiles

3. Programming in C
   - Comments and Documentation
   - C Data Types and Functions
   - The Preprocessor
A New Programming Language?

- Broaden your Experience
  - Look beyond Java
  - Ultimately: “Been there, done that”
- Get a feeling of “It’s easy”
  - Hard yards ahead, but eventually get rewarded
  - Syntax stumbling block becomes smaller
  - “They are all the same”
- Learn how to program (for real)
  - Needs lots of practice!
    - Learn from your own mistakes!
  - Don’t copy/paste or memorise!
- Divide a complex problem into simple parts
- Know were to look (and what to look for)
  - Programming Language reference
  - API reference
Why C?

- Most frequently used language
  - Tons of reusable code
- The Systems Programming language
  - Most Kernels are written in C
  - Insight into underlying concepts
- Procedural part of Objective-C and C++
- Predecessor of Java, C++, C#, Objective-C, ...
  - Very similar syntax
  - Concepts help you with these languages
  - But: no language concept of Classes and Objects!
Why Objective-C?

- **Object oriented additions to C**
  - Supports Classes and Objects (in addition to low level C)
  - Complex data types are easier to manage than in plain C

- **Object oriented additions are plain and simple**
  - Much simpler language than C++ and even Java
  - No burden from multiple inheritance, templates, operator overloading, etc.

- **Powerful, dynamic object concept**
  - Classes are first class objects
  - Fully dynamic dispatcher
  - Solid basis for OO concepts

- **Primary language for iPhone, iPod Touch, Mac OS X.**
Why C++?

- Object oriented additions to C
  - Supports Classes and Objects (in addition to low level C)
  - Complex data types are easier to manage than in plain C

- Lots of language additions over C
  - Templates, multiple inheritance, operator overloading
  - Powerful concepts in the right hands
    - But: easy to get it wrong!
    - Requires skilful programming
      - ⇒ hard to come by well-written C++ code

- Popular programming language
  - Still used heavily in industry
  - Used in 3622ICT Interactive Entertainment
C Overview – Core Properties

- Procedural Language
  - Global functions instead of Methods that are local to classes
- Low level language
  - Use of Pointers for references
  - “Assembly language in disguise”
  - Great for looking behind the scenes
- Standard C Library
  - Easy to write cross-platform (non-GUI) programs!
  - ANSI/ISO-C functions (supported everywhere)
    - Memory allocation, Input/output, string processing, mathematics, . . .
  - POSIX functions (supported almost everywhere)
    - Multitasking, networking, distributed computing, . . .
Hello World

Java

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

C

```c
int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

- **Function Definition** →
  - returns an `int` (0 for success)
  - `void` means “no parameters”
Migrating from Java to C

- Functions in C work like Methods in Java
  - take parameters
  - return values
  - are global (do not belong to objects)
- There can only be one global function with a given name
  - E.g., only one `main()` function
- In C the `main()` function returns `int`
  - `return 0` to indicate that your program was successful
printf()

- Print a formatted string
  - Standard C output function
- Prints to stdout
  - Normally on screen
  - Can be redirected into a file
- Takes a format string
  - More than just a simple string like “hello world”
  - Can take additional parameters
  - How these parameters are formatted is determined by place holders
Some Place Holders

%-s string, e.g. "Hello"
%-c single character, e.g. 'x'
%-d decimal signed integer, e.g. -2
%-u decimal unsigned integer, e.g. 5
%-f floating point value, e.g. 2.5
%-e exponent value, e.g. 2.5e3
%-g automatically formatted float, e.g. 2500.3
Place Holder Examples

- \texttt{printf("Hello, \%s", "world");}
  - Hello, world
- \texttt{printf("The distance is \%d km", 15);}
  - The distance is 15 km
- \texttt{printf("\%u times \%g is \%g", 3, 2.5, 3*2.5);}
  - 3 times 2.5 is 7.5
Place holders allow output formatting

The syntax is \% [-][0][n][.k][l]x

- left alignment (default: right)
- 0 leading zeros instead of spaces (numbers only)
- n minimum number of digits
- k cap at k digits maximum
- l long (e.g. long int)
- x The actual place holder character (s, d, f, etc.)

E.g.: \%5d decimal number with 5 digits
String Formatting Characters

- Work almost exactly as in Java!
  - \n  new line
  - \t  tabulator (indentation to the next multiple of 8)
  - \\  the backslash character \ itself
  - \"  double quote "
  - \‘  single quote ’
  - \0  end of string (ASCII 0)
  - \nnn  Character with octal value nnn
Putting it together

Example (What does this program print?)

```c
int main(void)
{
    int j;

    j = 7;
    printf("j = %03.3d\n", j);

    return 0;
}
```

Answer

```
j = 007
```
Compiling C Programs

- **Integrated Development Environment (IDE)**
  - Eclipse, XCode, Visual C++, Project Center, ...
  - Compiles programs at the press of a button (like BlueJ)
  - Often difficult to customise
  - Very rarely support multiple platforms and languages

- **Command Line**
  - Requires manual invocation
  - Requires knowledge of command line parameters
  - Can be tedious for large projects
  - Cross-platform and -language compilers (e.g. clang)

- **Makefiles**
  - Combine the best of both worlds
  - Recompile a complex project with a simple `make` command
Getting a Command Line Interface

- **Via Dwarf**
  - `ssh dwarf.ict.griffith.edu.au`
  - using putty (Windows)

- **Via a local Terminal**
  - Mac OS X: e.g. Applications / Utilities / Terminal.app
  - Linux: e.g. through the Gnome program menu
  - Windows: e.g. Start / Programs / Programming Tools / GNUstep / Shell

⇒ Enter commands to compile your program
- Hit *Return* (or *Enter*) after every command!
Compiling a C program using clang or gcc

- Once on the command line change to the directory (folder) your program is in
  - cd /my/example/directory
- Compile the source code (e.g. Hello.c)
  - clang Hello.c
  - Compiles Hello.c into an executable called a.out (or a.exe on Windows)
  - clang -o Hello Hello.c
  - Compiles Hello.c into an executable called Hello
  - On Windows always use Hello.exe instead of just Hello
- clang -Wall -o Hello Hello.c
  - Prints all warnings about possible problems
  - Always use -Wall when compiling your programs!
- ./Hello
  - Run the Hello command from the current directory
- To use gcc, simply replace clang with gcc
Makefiles

- Save compile time
  - only recompile what is necessary
- Help avoiding mistakes
  - prevent outdated modules from being linked together
- Language independent
  - work with any programming language
    - C, C++, Objective-C, Java, ...
How do Makefiles work?

Example (A simple Makefile)

Hello: Hello.c
    clang -Wall -o Hello Hello.c

- First Line: Dependency Tree
  - Target and Sources
  - Target: the module to be built (e.g. Hello)
  - Sources: pre-requisites (e.g. Hello.c)
Make Rules

Example (A simple Makefile)

Hello: Hello.c
    clang -Wall -o Hello Hello.c

- Second Line: Make rule
  - command to execute
    - clang -Wall -o Hello Hello.c
  - requires a **tab** character (not spaces) for indentation
Multiple Targets

Example (Makefile for compiling multiple Modules)

Program:  module1.o module2.o
  clang -o Program module1.o module2.o

module1.o:  module1.c
  clang -c -Wall -o module1.o module1.c

module2.o:  module2.c module2.h
  clang -c -Wall -o module2.o module2.c

- Default Target: first target (Program)
- link two object files (module1.o and module2.o) into one program (Program)
Multiple Targets (2)

Example (Makefile for compiling multiple Modules)

Program:  module1.o module2.o
          clang -o Program module1.o module2.o

module1.o:  module1.c
            clang -c -Wall -o module1.o module1.c

module2.o:  module2.c module2.h
            clang -c -Wall -o module2.o module2.c

- **Second Target:** module1.o
  - rule to compile object file module1.o from module1.c
  - `clang -c` compiles a single module (not a full executable)
Multiple Targets (3)

Example (Makefile for compiling multiple Modules)

Program:  module1.o module2.o
  clang -o Program module1.o module2.o

module1.o:  module1.c
  clang -c -Wall -o module1.o module1.c

module2.o:  module2.c module2.h
  clang -c -Wall -o module2.o module2.c

- Third Target: module2.o
  - compile module2.o from source module2.c
  - also depends on module2.h (header file)
Multiple Programs

Example (Makefile for compiling multiple Programs)

```
all: Program1 Program2

Program1: module1.o
    clang -o Program module1.o

Program2: module2.o module3.o
    clang -o Program module2.o module3.o

module1.o: module1.c
    clang -c -Wall -o module1.o module1.c

module2.o: module2.c module2.h
    clang -c -Wall -o module2.o module2.c

module3.o: module3.c module3.h
    clang -c -Wall -o module3.o module3.c
```

- **'all' target:**
  - compiles all programs (Program1 and Program2)
  - does not have any compiler commands itself!
Generic Rules

- Save lots of typing
  - avoid repeating the same compiler call over and over again
- Help with consistency
  - what if you want to change the compiler invocation?
- Simply list suffixes to convert from one file type to another
  - e.g. `.c` to compile a `.c` to a `.o` file
Generic Rule Example

Example (Makefile containing a generic rule)

```makefile
.c.o:
    clang -c -Wall -o $*.o $*.c

Program:  module1.o module2.o
    clang -o Program module1.o module2.o

module2.o:  module2.c module2.h
```

- `.c.o`:
  - how to compile a `.c` into a `.o` file
  - `$*` gets replaced by the file name (without extension)
Generic Rule Example (2)

Example (Makefile containing a generic rule)

```
.c.o:
    clang -c -Wall -o $*.o $*.c

Program:  module1.o module2.o
    clang -o Program module1.o module2.o

module2.o:  module2.c module2.h
```

- No need for a `module1.o: rule!`
- compiler already knows how to compile .c into .o
- But: `module2.o` needs a rule (also depends on .h)
Generic Rules for Languages other than C

- The `make` utility by default only knows about C
  - “what if I want to compile a different language?”
- Suffixes can be specified
  - using the `.SUFFIXES:` command, e.g.:
  - `.SUFFIXES: .o .m`
  - “a .o file can also be compiled from a .m (Objective-C) file”
Make Variables

- Allow more flexible make files
  - “what if the compiler is not called clang?”
- Variables allow assigning a value, e.g.:
  - CC=gcc
- Variables can be used using $(variable), e.g.:
  - $(CC) -c -Wall -o $*.o $*.c
  - will replace $(CC) with gcc
Mixed Makefile Example: Objective-C

Example (Makefile for a mixed C/Objective-C program)

# A mixed makefile example for C and Objective-C on Mac OS X
# CC=clang

.SUFFIXES: .o .c
.SUFFIXES: .o .m

.c.o:
   $(CC) -c -Wall -o $*o $*.c

.m.o:
   $(CC) -c -Wall -o $*o $*.m

Program: cmodule.o objcmodule.o
   $(CC) -o Program cmodule.o objcmodule.o -framework Foundation

objcmodule.o: objcmodule.m objcmodule.h
Mixed Makefile Example: C++

Example (Makefile for a mixed C/C++ program)

```
# A mixed makefile example for C and C++
CC=clang
CPLUS=g++

.SUFFIXES: .o .c
.SUFFIXES: .o .cc

.c.o:
    $(CC) -c -Wall -o $*.o $*.c

.cc.o:
    $(CPLUS) -c -Wall -o $*.o $*.cc

Program: cmodule.o cppmodule.o
        $(CPLUS) -o Program cmodule.o cppmodule.o

cppmodule.o: cppmodule.cc cppmodule.h
```
Comments

- Plain C allows comments between /* and */
  - /* this is a valid C comment */
- Comments may not be nested
  - /* this /* is not a valid C comment */ */
- C99 also allows double-slash // end-of-line comments
  - // this is a valid comment
  - no closing sequence needed – the comment ends at the end of the line
Comment Example

Example (Program with Comments)

```c
/*
 * This program prints "j = 007".
 * It does not take any parameters and returns 0 on success.
 */
int main(void) /* main function definition */
{
    int j; /* our int variable to play with */
    j = 7;  // assign a value to be printed
    printf("j = %03.3d\n", j);  // print value with leading zeroes
    return 0;  // everything is fine, exit program
}
```
Where to put comments?

- At the beginning of each file (module)
  - describe the name of the module, purpose, author, and dates when first created and last modified
- Before each function (method)
  - describe the purpose of the function or method,
  - input parameters (arguments),
  - return values (output parameters), and
  - pre- and postconditions (contract)
- At the beginning of each class
  - describe the purpose of the class, and
  - things to keep in mind when using this class
How to comment?

- Use comments to document important parts of your code
- Document key functionality
- Don’t re-iterate the obvious!

**Example (Bad comment)**

```c
i = 7;  // assign 7 to i
```

**Example (Better)**

```c
i = 7;  // seven iterations to go
```
Everybody hates writing documentation, right?
- can be lots of work
- duplicated efforts if all the information is already in the source code

The good news: Tools that extract documentation from the source
- JavaDoc (Java specific)
  - in the labs: can use JavaDoc syntax for C, C++, Objective-C
- Doxygen (http://www.stack.nl/~dimitri/doxygen/)
  - similar, installed on dwarf
- AutoGSDoc
  - part of the GNUstep environment on Linux and Windows
Example

```c
/**
 * The main() function of this program prints "Hello World" and
 * then exits. This function does not take any parameters and
 * returns 0 to indicate success.
 */

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```
C Statements

- C Statements use the same Syntax as in Java
- There is only a small number of keywords
- Let’s have a look at some of them!
  - Operators: +, −, ++, ...
  - Conditionals: if, case, ?
  - Loops: do, while, for
  - Control: return, break, continue, ...
# Increment/Decrement Example

**Example (What does this program print?)**

```c
int main(void)
{
    int x = 5;

    --x; // pre-decrement
    printf("x = %d\n", x++); // post-increment

    printf("x now is %d\n", x); // print final result

    return 0;
}
```

**Answer**

- `x = 4`
- `x now is 5`
Example (What does this program print?)

```c
int main(void)
{
    int x = 5;

    if (x < 2) // check if x < 2
    {
        printf("%d < 2\n", x);
    }
    else
    {
        printf("%d >= 2\n", x);
    }

    return 0;
}
```

Answer

5 >= 2
Case Statement Example

Example (What does this program print?)

```c
int main(void)
{
    int x = 5;

    switch (x) // let's see what the value of x is
    {
        case 0:
            printf("x is zero\n");
            break;
        case 5:
            printf("x is five\n");
        default:
            printf("x is %d\n", x);
    }

    return 0;
}
```

Answer

- x is five
- x is 5
Question Mark Operator Example

Example (What does this program print?)

```c
int main(void)
{
    int x = 5;

    printf("x is %s\n", x > 2 ? "greater than 2": "not greater 2");

    return 0;
}
```

Answer

`x is greater than 2`
Example (What does this program print?)

```c
int main(void)
{
    int x = 5, y = 0, i;  // declare multiple variables

    for (i = 0; i < x; i++)  // a 'for' loop
    {
        y += i;  // add i to y
    }

    printf("y = %d\n", y);  // print the result

    return 0;
}
```

Answer

\( y = 10 \)
While Loop Example

Example (What does this program print – are you sure?)

```c
int main(void)
{
    int x = 5, y = 0, i = 5;  // declare some variables

    while (i < x)  // a 'while' loop
    {
        y += i++;  // add i to y, then increment i
    }

    printf("y = %d\n", y);  // print the result

    return 0;
}
```

Answer

\[ y = 0 \]
Example (What does this program print?)

```c
int main(void)
{
    int x = 5, y = 0, i = 5;    // declare some variables
    do
    {
        y += i++;                  // add i to y, then increment i
    } while (i < x);

    printf("y = %d\n", y);   // print the result
    return 0;
}
```

Answer

\[ y = 5 \]
Example (What does this program print?)

```c
int main(void) {
    int x = 5, y = 0, i = 0; // declare some variables

    while (i++ < x) {
        if (i == 1) continue; // continue if i is 1
        if (i == 3) break; // break if i is 3
        y += i;
    }

    printf("y = %d\n", y); // print the result

    return 0;
}
```

Answer

```
y = 2
```
Primitive data types

- **Primitive data types**
  - **char, short, int, long, long long**
  - Integer data types (e.g. 5 or -7)
  - Can be prefixed with `unsigned` (no negative numbers) or `signed`
  - Sizes are compiler specific (e.g. 4 bytes for an `int`), but:
  - `char ≤ short ≤ int ≤ long ≤ long long`
  - Unless `unsigned` is specified, all types (except `char`) are always `signed`
  - Whether `char` is `unsigned` or `signed` by default is compiler specific
- **float, double, long double**
  - Floating point (real) numbers
  - E.g. 3.5, -7.2e4
### Example (What does this program print?)

```c
int main(void)
{
    int x = -2; // a signed integer variable
    unsigned y = 3; // an unsigned integer variable
    float f = 2.5; // a floating point variable
    double r, d = -2.5e3; // two double variables ‘r’ and ‘d’

    r = d / f * x + y; // let’s do some maths
    printf("r = %lg\n", r); // and print the result

    return 0; // exit main() and report success
}
```

### Answer

\( r = 2003 \)
Square brackets \([\ ]\) denote a fixed-size array
The size of an array is static and cannot be changed!

**Example**

- \(\text{long } a[5]\)  
  array of 5 long integers
- \(\text{unsigned } u[4]\)  
  array of 4 unsigned integers
- \(\text{unsigned long } u[4]\)  
  array of 4 unsigned, long integers
- \(\text{float } v[3]\)  
  array of 3 floats (e.g. a 3D vector)
- \(\text{double } m[4][5]\)  
  a 2-dimensional 4-by-5 array (matrix) of doubles
- \(\text{char } s[5]\)  
  array of 5 characters
  \(\Rightarrow\) a string!
Example (What does this program print?)

```c
int main(void)
{
    char s1[5] = "to C";       // a string 's1' of five characters
    char s2[8] = "Welcome";   // a string 's2' of eight characters

    printf("%s %s\n", s2, s1);  // print s2 followed by s1

    return 0;                  // exit main() and report success
}
```

Answer

Welcome to C

Does anyone notice anything strange? Each string has an invisible character `\0` at the end to denote the end of the string! Strings need space for one more character in addition to their length!
Indexing Arrays

Example (What does this program print?)

```c
int main(void)
{
    int years[3]; // an array of three ints
    years[0] = 2006; // first element
    years[1] = 2007; // second element
    years[2] = 2008; // third element
    int year = years[2]; // pick element at index two
    printf("The year is %d\n", year); // print the year
    return 0;
}
```

Answer

The year is 2008
### Static Initialisation of Arrays

**Example (What does this program print?)**

```c
int main(void)
{
    int i = 1;          // array index to use
    int year = years[i]; // pick the year with index 'i'

    printf("The year at index %d is %d\n", i, year);

    return 0;
}
```

**Answer**

The year at index 1 is 2007
Properties of C Arrays

- Multiple elements of the same kind
  - laid out contiguously in memory
- Can contain any data type
- Fixed (maximum) size
- Single or multi dimensional
Compound Data Types

- Structures, Unions, and Bit Fields
- Can contain multiple different data types
- Look very similar to classes in Java
- Member variables, but no methods!
Example (What does this program print?)

```c
struct Profit // definition of a 'Profit' structure
{
    int year; // the year this profit is reported for
    double dollars; // the actual profit in dollars
};

int main(void) // here starts the actual program (main)
{
    struct Profit myProfit; // a 'myProfit' variable of type 'Profit'

    myProfit.year = 2007; // myProfit is for the year 2007
    myProfit.dollars = 1234.5; // with a bottom line of 1234.5 dollars

    printf("In %d, I made %g dollars\n", myProfit.year, myProfit.dollars);

    return 0;
}
```

Answer

In 2007, I made 1234.5 dollars
Static Initialisation of Structures

Example (What does this program print?)

```c
struct Profit // last example’s ‘Profit’ structure
{
    int year;
    double dollars;
};

int main(void)
{
    /*
    * initialise myProfit statically
    */
    struct Profit myProfit = { 2007, 1234.5 };  

    printf("In %d, I made %g dollars\n", myProfit.year, myProfit.dollars);

    return 0;
}
```

Answer

In 2007, I made 1234.5 dollars
C Functions

- Similar to Java methods
  - syntax for parameters and return values is the same
- Functions are global rather than local
  - no two global functions can have the same name!
Example (What does this program print?)

```c
/**
 * a simple calc function that takes a signed and an unsigned integer
 * and returns a double
 */
double calc(int x, unsigned y)
{
float f = 2.5;
double d = 2.5e3;

return d / f * x + y;  // let’s do some maths
}

int main(void)
{
double r = calc(2, 3);  // invoke calc and store result in r

printf("r = %lg\n", r);  // and print the result

return 0;  // exit main() and report success
}
```

Answer

r = 2003
Function Declarations

- C Compiler only knows code it has already seen
- Functions need to be declared for the compiler to know them
- Forward declarations allow function calls before the actual function gets defined
- Syntax: function header followed by ; – e.g.:
  - `int main(void);`
  - `int myFunction(int, double);`
  - `double average(double, double);`
Function Declaration Example

Example

```c
double calc(int, unsigned); // function declaration of calc()

int main(void)
{
    double r = calc(2, 3); // invoke calc and store result in r

    printf("r = %lg\n", r); // print the result

    return 0;
}

double calc(int x, unsigned y) // the actual calc() function
{
    float f = 2.5;
    double d = 2.5e3;

    return d / f * x + y;
}
```

René Hexel

An Introduction to C
Example (What is wrong with this program?)

```c
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

Answer
Hello.c: In function ‘main’: Hello.c:3: warning: implicit declaration of function ‘printf’
Hello.c:3: warning: incompatible implicit declaration of built-in function ‘printf’
Error-Free version of “Hello World”

Example (Using the Pre-processor)

```c
#include <stdio.h>     // include the declaration for printf()

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

- `#include <...>` includes a header file
- `#include <stdio.h>` includes the relevant declaration for `printf()`
- `similar functionality to “import” in Java`
#include

- Includes global or local header files
  - `#include <stdio.h>`  // include a global header file
  - `#include "hello.h"`  // include a local header file

- Header files are just files that get inserted instead of the `#include` statement
  - could be any C code
  - by convention, only contains declarations but no definitions!
  - use a `.h` extension

- API defines a set of standard header files (include files)
ISO C Standard Include Files

- `#include <stdio.h>`
  - Standard Input/Output header
  - `printf()` for formatted output
  - `scanf()` for formatted input, ...

- `#include <string.h>`
  - String functions

- `#include <math.h>`
  - Mathematics functions

- `#include <stdlib.h>`
  - Memory management, data conversion, `exit()`, etc.

- Defined in the C Language standard
  - [http://std.dkuug.dk/JTC1/SC22/WG14/www/docs/n843.htm](http://std.dkuug.dk/JTC1/SC22/WG14/www/docs/n843.htm) (draft, section 7.1.2 and Annex B)

- API defines a set of standard header files (include files)
#define Macros

- All the C preprocessor does is text replacement
  - before the actual compiler kicks in
  - but it is very good at that!
- \#define \textbf{A} \textbf{B}
  - replaces \textit{A} with \textit{B} in the code
  - \textit{A} and \textit{B} can be complex text
- e.g. Constants
  - \#define EXIT_SUCCESS 0
    - replaces EXIT_SUCCESS with 0 in the code
    - e.g. \texttt{return EXIT_SUCCESS; instead of return 0;}
    - the purpose of the \texttt{return} statement is explained in code
    - makes the code more readable
More sophisticated than simple constants

```c
#define ERROR(x) printf("Error %d\n", x)
```

- replaces `ERROR()` with the complex `printf()` statement
- replaces `x` with the text parameter given to `ERROR()`
- `ERROR(5);` gets translated to
  ```c
  printf("Error %d\n", 5)
  ```
- prints “Error 5”
#define is very powerful

- Lets you replace functions with macros
  - can increase code readability
  - can increase code efficiency
  - can reduce errors for repetitive code sequences
    - use macros instead of copy/paste
  - Don’t overdo it!
    - use functions/methods where you can
    - only use macros where it increases code readability

Beware of side effects!

- Macro invocations are not actual method invocations!
- All #define does it text replacement!
Example (What does this code print?)

```c
#define TRIPLE(x) printf("%d * 3 = %d", x, x * 3)

int j = 4;
TRIPLE(j + 1);  // prints "5 * 3 = 15", right?
```

Answer

5 * 3 = 7

Because `TRIPLE(i + 1)` gets expanded to:

```c
printf("%d * 3 = %d", i + 1, i + 1 * 3);
```

Important

⇒ Always put Macro arguments in brackets ()

- e.g. `#define TRIPLE(x) printf("%d * 3 = %d", (x), (x) * 3);`
Side Effect Example

Example (Do the brackets help here?)

```c
#define TEST(x) if ((x) < 0) printf("%d < 0\n", (x))

int i = -1;
TEST(i++); // what does this statement yield?
printf("i = %d\n", i); // what does this statement print?
```

Answer

```
0 < 0
i = 1
```

Why?

Because `TEST(i++)` gets expanded to:

```
if ((i++) < 0) printf("%d < 0\n", (i++));
```