Two languages for the price of one

- Before being passed to the compiler proper, your program passes through a preprocessor.
  - Your program is passed first to the preprocessor, and the result is further passed to the C++ compiler.
- The preprocessor has a language of its own.
  - This language is not part of C++.
  - In particular, it has a different syntax, and requires a different mindset to use.
  - Most problems occur when the preprocessor is treated like C++.
  - The preprocessor language is tailored to the task of translating code.
- In a C++ program, you should not abuse the preprocessor.
  - Use it when needed.
  - Use it to increase efficiency, but only if you can think of no alternative (and keep in mind that such increased efficiency is often not justified).

Include directives

- The functionality of the preprocessor is based on directives.
- A preprocessor directive starts with a # character and extends to the end of line.
  - There is no terminating semicolon.
- A useful directive:

```
#include <iostream>
#include "lists.h"
```

- The effect of `#include "foo.h"` is the replacement of the directive with the content of the file “foo.h”.
  - Filenames can be passed to `#include` using an absolute (e.g., `/usr/include/stdio.h`) or relative (e.g., `sys/stat.h`) path.
    - Under Windows, you should use the backslash (`\`) instead of slash (`/`).
    - As opposed to C++ proper, do not use `\\`!
  - Relative to what?

Include directives

- The functionality of the preprocessor is based on directives.
- A preprocessor directive starts with a # character and extends to the end of line.
  - There is no terminating semicolon.
- A useful directive:

```
#include <iostream>
#include "lists.h"
```

- The effect of `#include "foo.h"` is the replacement of the directive with the content of the file “foo.h”.
  - Filenames can be passed to `#include` using an absolute (e.g., `/usr/include/stdio.h`) or relative (e.g., `sys/stat.h`) path.
    - Under Windows, you should use the backslash (`\`) instead of slash (`/`).
    - As opposed to C++ proper, do not use `\\`!
  - Relative to what?

    - To predefined directories with known headers, and to the current directory (`.`).

Include directives (cont’d)

- There are two variants of an include directive.

```
#include <iostream>
#include "lists.h"
```

- The difference is the order in which the directories are searched for the respective file.
  - The angle bracketed version causes the preprocessor to look into the predefined directories first.
  - The double quoted variant tells the preprocessor to look first in the current directory.
  - The latter is normally used to include the headers written by you.
  - Proper use of these variants is a matter of self-documentation of the code, and is thus encouraged.
- The `#include` directive is intended for inclusion of header files. Using it like this:

```
#include "btree.cc"
```

  is certainly possible, but is very bad programming practice. (why?)

Conditional compilation

- Problem. We want to build a program that compiles under Windows as well as Unix. What do we do with the #include directives?
- Solution. We use conditional compilation:

```
#ifndef __MSDOS__
#include <sys/stat.h>
#else /* __MSDOS__ */
#include <sys\stat.h>
#endif /* __MSDOS__ */

#ifdef __MSDOS__
const char* filename = "\\home\\bruda\\foo";
#else /* __MSDOS__ */
const char* filename = "/home/bruda/foo";
#endif /* __MSDOS__ */
```

- The portion of the file between #ifdef C and #endif is passed to the compiler if and only if the “macro” C is defined using #define.
  - Some macros are defined for you, and you can define more using #define in your program or the -D switch of g++.

Conditional compilation (cont’d)

- Another example of conditional compilation: debug code

```
#ifdef DEBUG
cout << "### added " << lst -> car << " to " << lst << "\n";
#endif /* DEBUG */
```

- Whenever you want to debug your program, you can define DEBUG as follows:
  - In the code of the module you need to debug, by putting the following directive at the beginning of the C++ file

```
#define DEBUG
```

  - If your module is called foo, you can define DEBUG for it at compile time:

```
g++ -g -Wall -DDEBUG -o foo.o foo.cc
```

- You can also “undefine” a macro:

```
#undef DEBUG
```

More define directives

- We defined up to this point macros without values.
  - I.e., they either exist or not.
  - Useful for conditional compilation.
- We can also associate values with our macros.

```
#define SIZE 128
```

In general, we write: #define *Name* *Substitute-text*

  - The effect: the string *Name* is literally and globally replaced with the string *Substitute-text* throughout the code before the code is passed to the C++ compiler.

Macros versus const variables

- Compare:

```
#define SIZE 128
const int SIZE = 128;
```

- const variables are preferred over macros.
  - A variable declaration uses familiar syntax.
  - The syntax of a variable declaration is checked immediately.
    - ∗ The syntax of a #define directive is checked when it is first used.
    - ∗ The error line reported by the compiler is not the line where the error actually happens!
  - A variable declaration follows scoping rules; a #define directive is always global.
  - It might be the case that a macro produces more efficient code, but the efficiency gain is negligible for most normal programs.
- Sometimes, however you are better off if you use macros.
  - How would you define the constant NULL?

Unexpected results

```
1.  // Real error on line 2:
2.  #define BIG_NUMBER 10 ** 10
3.
4.  int main () {
5.    int i = 0;
6.    while ( i < BIG_NUMBER )  // Error signalled on line 6!
7.      i *= 10;
8.  }
```

```
#define A_NUM 7
#define ANOTHER_NUM 6
#define A_SUM A_NUM + ANOTHER_NUM

cout << "Squared sum: " << A_SUM * A_SUM << "\n";
```

```
#define MAX =10

for (counter=MAX; counter > 0; counter --)  // error and warning here!
  cout << "Hello\n";
```

Things you can but should not do with macros

- Obscure the basic control flow of a program:

```
#define FOR_ALL for (int i = 0; i < ARRAY_SIZE; i++)

FOR_ALL {
  data[i] = 0;
}
```

- Obfuscate your code, e.g., by using a half-C++, half-Pascal language:

```
#define begin {
#define end }

if (index == 0)
begin
  data[i] = -1;
end
```

Parameterized macros

- Macros can also take parameters:

```
#define SQR(x) ((x) * (x))
#define MAX(x,y) ( (x) < (y) ? (y) : (x) )
#define RECIP (x) ( 1.0 / (x) )


for (int i = 0; i < 10; i++)
  cout << SQR(i);
cout << MAX(1,2) << " " << RECIP(1);  // undefined variable x!
```

Parameterized macros

- Macros can also take parameters:

```
#define SQR(x) ((x) * (x))
#define MAX(x,y) ( (x) < (y) ? (y) : (x) )
#define RECIP(x) ( 1.0 / (x) )


for (int i = 0; i < 10; i++)
  cout << SQR(i);
cout << MAX(1,2) << " " << RECIP(1);
```

- Never put inside parameterized, arithmetic macros operations with side effects (such as ++).
  - In other words, differentiate between macros that do arithmetic and macros that contain statements, and never mix them.
- Do not separate the list of parameters from the name of the macro.

RULES TO MINIMIZE TROUBLES

- Macros are sometimes unavoidable and/or make your life easier. But they tend to create trouble if you abuse them and/or you make mistakes when defining them.
- When working with macros, KISS (keep it simple, stupid).
  - define empty macro as you need them
  - define parameterless macros if you cannot think of anything else
  - think twice before declaring macros with parameters.
- Put brackets around everything in an arithmetic macro.
- When defining a macro with more than one C++ statement, surround it by braces.
- The preprocessor is not C++. Do not use C++ syntax.