Outline

- Basic Inline Assembly
- Extended Assembly
- The C Preprocessor
AS10-Inline Assembly

Introduction

You may want to write portions of a C++ application in assembly for increased speed or to access special instructions (BCD arithmetic, MMX, SSE, etc).

You have three options:

- Write a complete function in assembly from scratch and call it from C++.
- Write a function or program in C++, translate to assembly and then modify the assembly.
- Mix assembly with C++ using *inline* assembly.
Assembly can be inserted into a GNU C++ program using the `asm` keyword:

```c
asm("movl $100, %eax
	" "addl $200, %eax
	" "call print_int");
```

The instructions must be enclosed in quotation marks and newline characters must be used to separate lines of assembly. The tab characters will indent the generated assembly for readability.
You must take care not to alter the values of any registers in your assembly code. This is typically achieved with a pusha/popa pair.

It is relatively simple to access C++ global variables from assembly. They are accessed in exactly the same way as assembly labels in the data or bss sections. Remember to prepend an underscore to variable names in Cygwin (it is not necessary in Linux).
Here is a complete C++ example:

```cpp
int a=10; int b=20; int result;
int main() {
    asm("pushl %eax\n\t"
         "movl _a,%eax\n\t"
         "imull _b,%eax\n\t"
         "movl %eax,_result\n\t"
         "popl %eax");
    cout << a << "\t" << b \\
        << "\t" = " \t" << result << endl;
}
```
The following code would copy the CPUID string to a global `char buffer[13]`:

```asm
asm("pusha\n\t"
 "movl  $0,\%eax\n\t"
 "cpuid\n\t"
 "movl  \_buffer,\%eax\n\t"
 "movl  \%ebx,  (\%eax)\n\t"
 "movl  \%edx,  4(\%eax)\n\t"
 "movl  \%ecx,  8(\%eax)\n\t"
 "movb  $0,  13(\%eax)\n\t"
 "popa") ;
```
g++ provides **extended assembly** which allows access to **local** variables and can automatically save and restore registers.

The syntax for extended **asm** is:

```asm
asm("asm code":
    output locations:
    input operands:
    changed registers);
```
The **asm code** is the same as for basic asm.

The **output locations** is a list of registers and memory locations that contain output values from the asm code.

The **input operands** is a list of registers and memory locations that contain input values for the asm code.

The **changed registers** is a list of additional registers that are changed by the asm code.
The format of items in the input and output list are of the form:

“constraint”(variable)

where constraint is a letter indicating where the variable is copied to/from (input/output).

Letters \texttt{a}, \texttt{b}, \texttt{c}, and \texttt{d} indicate the EAX, EBX, ECX, and EDX registers (or portions thereof depending on the size of the variable). \texttt{S} and \texttt{D} indicate ESI and EDI.
The letter \texttt{r} indicates any available register. The letter \texttt{m} indicates that the variable should be accessed from memory. See pg 371 for the complete list of constraints.

An output constraint can be modified with either a + to indicate that the variable is used for both input and output or an = to indicate that the variable can only be written to.
Here is an example of extended assembly:

```c++
int a = 10, b = 20;
int result;
asm("imull %%ecx,%%eax"
     : "=a"(result)
     : "a"(a), "c"(b));

cout << a << "\times" << b << " = "
    << result << endl;
```
Another example:

```cpp
char *buffer = new char[13];
asm("cpuid\n\t" "movl %ebx, \(%%esi)\n\t" "movl %edx, 4(\(%%esi)\n\t" "movl %ecx, 8(\(%%esi)\n\t" "movb $0, 13(\(%%esi)"
::"a"(0), "S"(buffer)
: "%ebx", "%ecx", "%edx"));
cout << buffer << endl;
```
In the previous example, an input variable specification is used to move a zero into the EAX register. This could have been done in the assembly code. This example also uses a changed registers lists.

Since there is no output variable, the compiler might delete the assembly code in an optimization step. This can be prevented by using the `volatile` keyword:

```c
asm volatile ("code",...);
```
Note that register names are prefixed with %%% instead of % in the assembly code (but not in the changed registers list). A single % is used to indicate a placeholder.

There is much more to extended assembly (placeholders, FPU registers, using memory references), but that is all that we will cover here. See the text or the gcc info page for more information.
If the assembly file has a .S (capital S) extension the file will automatically be run through the C preprocessor before assembly. This allows us to use preprocessor macro definitions in our assembly programs:

```c
#define NBYTES 100
#define XVAL   8(%ebp)
#define SUM    -4(%ebp)
#define COUNT  %ecx
```
The definitions can then be used in our assembly code as follows:

```
enter $NBYTES, 0
movl XVAL, %edx
addl SUM, COUNT
```

When using the C preprocessor, you should use C++ style comments (// or /* */) instead of #. The # is used as a prefix to the C preprocessor directives.
Although the C preprocessor can be used to write more readable source code, g++ cannot be invoked with both the debug option (-g) and assembly that needs to be preprocessed. You will need to break the compilation into two steps:

```sh
cpp assem.S output.s

g++ -g -o foo driver.cpp output.s
```