**DIFFERENCES BETWEEN** C **AND** C++

#include **Directives**

1. C++ directives

* Library functions, variables, and constants are made available to a program by specifying the name of header file using an #include directive.
* Example

#include <iostream>

#include <string>

* The built-in library routines (like cout and cin) and data types (like string) are used so frequently and commonly that we add

using namespace std;

at the top of our source code to simplify the references to these routines.

1. C directives

* Library functions, variables, and constants are made available to a program by specifying the name of header file using an #include directive.
* Example

#include <stdio.h>

#include <string.h>

* C does not use namespaces, so there is no need for

using namespace std;

at the top of our source code.

1. Many C libraries serve the same purpose as those in C++, but of course, the names are different.

* Example

#include <string.h> vs #include <string>

#include <stdlib.h> vs #include <cstdlib>

#include <math.h> vs #include <cmath>

**Comments**

1. C++ comments

* Indicated by two forward slashes (i.e., //) occurring anywhere on a line.
* All characters from // to the end of the line will be ignored.
* Example

// This is a single line comment

* Example

// This is the first line of

// a multi-line comment

* Example

if (some\_expression) // This might need a comment

* Example

// If you have a long comment, you can add a single backward slash \

to the end of the line

1. C comments

* The beginning of a comment is indicated by a forward slash followed by an asterisk (i.e., /\*).
* The end of a comment is indicated by an asterisk followed by a forward slash (i.e., \*/).
* Example

/\* This is a single line comment \*/

* Example

/\*

The single line comment above could have been done like this

\*/

* Example

/\* This is the first line of

a multi-line comment \*/

* Example

/\*

The multi-line comment above could have

been done like this

\*/

1. Combining C++ and C comments

* Multi-line C comments can be combined with C++ comments to easily comment and uncomment multiple lines of code.
* Example (the following code will be ignored by the compiler)

/\*

some line of code 1

some line of code 2

.

.

.

some line of code n

//\*/

* Example (the following code will be compiled)

//\*

some line of code 1

some line of code 2

.

.

.

some line of code n

//\*/

**Input and Output**

1. C++

* cin: Standard input
* cout: Standard output
* cerr: Standard error

1. C

* stdin: Standard input
* stdout: Standard output
* stderr: Standard error
* stdio.h provides printf, fprintf, sprintf, fputs, scanf, fscanf, and fgets functions.
* The most common output function is printf. It prints to wherever stdout is directed.
* Example

#include <stdio.h>

int main ()

{

printf (“Hello world\n”);

return 0;

}

* A printf function call actually consists of a format string followed by a variable number of arguments. The format string contains ordinary characters and format specifications. Format specifications are placeholders, one for each argument following the format string, describing how each of the corresponding arguments should be formatted.
* Example

#include <stdio.h>

int main ()

{

int i;

for (i = 1; i <= 10; i ++)

{

printf (“Loop pass = %d\n”, i);

}

return 0;

}

The result is:

Loop pass = 1

.

.

.

Loop pass = 10

* There are many format specifications, but some of the more commonly used ones are given below:

*Format Data Type Prints*

%d (or %i) int signed integer

%f float or double signed decimal

%s char \* C-string

%x int hexadecimal

%p void \* pointer

* If you want more control over printing, use the fprintf function. The first and second arguments of a printf function call are the second and third arguments to an fprintf function call. The first argument to an fprintf function call is the destination stream (i.e., stdout or stderr).
* Example

#include <stdio.h>

int main ()

{

int i;

for (i = 1; i <= 10; i ++)

{

fprintf (stdout, “Loop pass = %d\n”, i);

}

return 0;

}

* Another common output function is fputs. The first and second arguments of an fputs function call are a C-string and the destination stream, respectively.
* Example

#include <stdio.h>

int main ()

{

int i;

for (i = 1; i <= 10; i ++)

{

fputs (“Loop pass = “, stdout);

fprintf (stdout, “%d\n”, i);

}

return 0;

}

* The scanf input function reads formatted input. It reads from wherever stdin is directed.
* Example

#include <stdio.h>

int main ()

{

int i;

float f;

   char c;

   char s [100];

   printf (“Enter an integer: ”);

scanf (“%d”, &i);

printf (“Enter a float: ”);

scanf (“%f”, &f);

printf (“Enter a character: ”);

   scanf(“%c”, &c);

printf (“Enter a string: ”);

scanf (“%s”, &s);

printf (“\nInteger = %d\n”, i);

printf (“Float = %f\n”, f);

   printf (“Character = %c\n”, c);

   printf (“String = %s\n”, s);

return 0;

}

Assuming the following interactive session:

Enter an integer: 1

Enter a float: 2.5

Enter a character: x

Enter a string: text

The result is:

Integer = 1

Float = 2.5

Character = x

String = text

* The fgets input function reads characters into a C-string. It reads from wherever stdin is directed.
* Example

#include <stdio.h>

int main ()

{

char \*sPtr;

   char s [10];

   printf (“Enter a string: ”);

sPtr = fgets (s, 10, stdin);

if (sPtr != NULL)

{

   printf (“\nString = %s\n”, sPtr);

}

return 0;

}

Assuming the following interactive session:

Enter a string: abcdefghijklmnopqrstuvwxyz

The result is:

String = abcdefdgh

* Example

#include <stdio.h>

int main ()

{

char \*sPtr;

   char s [100];

   printf (“Enter a string: ”);

if (sPtr = fgets (s, 100, stdin))

{

while (\*sPtr == ‘ ’ || \*sPtr == ‘\t’)

{

sPtr ++;

}

}

printf (“\nString = %s\n”, sPtr);

return 0;

}

Assuming the following interactive session:

Enter a string: abcdefghijklm nopqrstuvwxyz

The result is:

String = abcdefghijklm nopqrstuvwxyz

**Passing Arguments**

1. C++

* Default for passing arguments is pass-by-value.
* Example

#include <iostream>

using namespace std;

void swap (int x, int y)

{

int temp;

temp = x;

x = y;

y = temp;

return;

}

int main ()

{

int a = 100;

int b = 200;

cout << “Before swap:” << endl;

cout << “ a = ” << a << endl;

cout << “ b = ” << b << endl;

swap (a, b);

cout << endl;

cout << “After swap:” << endl;

cout << “ a = ” << a << endl;

cout << “ b = ” << b << endl;

return 0;

}

The result is:

Before swap:

a = 100

b = 200

After swap:

a = 100

b = 200

* Also allows pass-by-reference.
* Example

#include <iostream>

using namespace std;

void swap (int &x, int &y)

{

int temp;

temp = x;

x = y;

y = temp;

return;

}

int main ()

{

int a = 100;

int b = 200;

cout << “Before swap:” << endl;

cout << “ a = ” << a << endl;

cout << “ b = ” << b << endl;

swap (a, b);

cout << endl;

cout << “After swap:” << endl;

cout << “ a = ” << a << endl;

cout << “ b = ” << b << endl;

return 0;

}

The result is:

Before swap:

a = 100

b = 200

After swap:

a = 200

b = 100

1. C

* All arguments are pass-by-value, there is no pass-by-reference mechanism. In order to simulate the pass-by-reference mechanism, pass the address of the variable whose value you want to change in the function, then reference the address as a pointer inside the function.
* Example

#include <stdio.h>

void swap (int \*x, int \*y)

{

int temp;

temp = \*x;

\*x = \*y;

\*y = temp;

return;

}

int main ()

{

int a = 100;

int b = 200;

printf (“Before swap:\n”);

printf (“ a = %d\n”, a);

printf (“ b = %d\n”, b);

swap (&a, &b);

printf (“\n”);

printf (“After swap:\n”);

printf (“ a = %d\n”, a);

printf (“ b = %d\n”, b);

return 0;

}

The result is:

Before swap:

a = 100

b = 200

After swap:

a = 200

b = 100

**Memory Management**

1. C++

* Uses new and delete.

1. C

* All memory allocation is done with the malloc function. The malloc function returns a void pointer to the memory allocated.
* The malloc function can be used to dynamically allocate built-in data types.
* Example

#include <stdio.h>

#include <stdlib.h>

int main ()

{

int \*iPtr;

   iPtr = (int \*) malloc (sizeof (int));

if (iPtr == NULL)

{

printf (“Memory allocation problem!\n”);

exit (1);

}

\*iPtr = 5;

printf (“\niPtr = %p\n”, iPtr);

printf (“\*iPtr = %d\n”, \*iPtr);

return 0;

}

Assuming the integer was allocated at memory location 0x118b010, the result is:

iPtr = 0x118b010

\*iPtr = 5

* The malloc function can be used to dynamically allocate arrays.
* Example

#include <stdio.h>

#include <stdlib.h>

int main ()

{

int \*iPtr;

iPtr = (int \*) malloc (sizeof (int) \* 100);

if (iPtr == NULL)

{

printf (“Memory allocation problem!\n”);

exit (1);

}

iPtr [0] = 99;

iPtr [99] = 0;

printf (“\niPtr = %p\n”, iPtr);

printf (“&(iPtr [0]) = %p\n”, &(iPtr [0]));

printf (“&(iPtr [99]) = %p\n”, &(iPtr [99]));

printf (“\niPtr [0] = %d\n”, iPtr [0]);

printf (“\*(&(iPtr [0])) = %d\n”, \*(&(iPtr [0])));

printf (“iPtr [99] = %d\n”, iPtr [99]);

printf (“\*(&(iPtr [99])) = %d\n”, \*(&(iPtr [99])));

\*iPtr = 98;

\*(iPtr + 99) = 1;

printf (“\n\*iPtr = %d\n”, \*iPtr);

printf (“\*(iPtr + 99) = %d\n”, \*(iPtr + 99));

return 0;

}

Assuming the array was allocated at memory location 0x2524010, the result is:

iPtr = 0x2524010

&(iPtr [0]) = 0x2524010

&(iPtr [99]) = 0x252419c

iPtr [0] = 99

\*(&(iPtr [0])) = 99

iPtr [99] = 0

\*(&(iPtr [99])) = 0

\*iPtr = 98

\*(iPtr + 99) = 1

* The malloc function can be used to dynamically allocate user-defined types.
* Example

#include <stdio.h>

#include <stdlib.h>

struct Point

{

int x;

int y;

};

int main ()

{

struct Point \*pPtr;

   pPtr = (struct Point \*) malloc (sizeof (struct Point));

if (pPtr == NULL)

{

printf (“Memory allocation problem!\n”);

exit (1);

}

pPtr -> x = 5;

pPtr -> y = 25;

printf (“\npPtr = %p\n”, pPtr);

printf (“pPtr -> x = %d\n”, pPtr -> x);

printf (“pPtr -> y = %d\n”, pPtr -> y);

return 0;

}

Assuming the pointer was allocated at memory location 0xe7d010, the result is:

pPtr = 0xe7d010

pPtr -> x = 5

pPtr -> y = 25

**Boolean Variables**

1. C++

* There is a built-in bool data type.

1. C

* Not all versions have a built-in bool data type. Instead the int data type is used to simulate a Boolean variable, where the value 0 represents false and any non-zero value represents true.
* Example

#include <stdio.h>

int main ()

{

int true = 5;

int false = 0;

printf (“\n”);

while (true)

{

printf (“true = %d\n”, true);

true --;

}

printf (“\ntrue = %d\n”, true);

if (!false)

{

printf (“\nfalse = %d\n”, false);

}

return 0;

}

The result is:

true = 5

true = 4

true = 3

true = 2

true = 1

true = 0

false = 0

* Some versions have a library called stdbool.h that has support for a built-in bool data type.
* Example

#include <stdio.h>

#include <stdbool.h>

int main ()

{

printf (“\ntrue = %d\n”, true);

printf (“false = %d\n”, false);

return 0;

}

The result is:

true = 1

false = 0