C/C++ Language Reference

Summary

Technical Reference
TR0173 (v4.0) April 06, 2009

This comprehensive reference provides a detailed overview of the C/C++ language and describes each of the standard C/C++ keywords (reserved words) and each of the standard C and C++ library functions. In addition, processor specific keywords and intrinsic functions are described. These do not belong to the standard C language, but are supported by the compilers for certain processors.

C/C++ Operator Precedence

The operators at the top of this list are evaluated first.

Precedence	Operator	Description	Example	Associativity
1	::	Scoping operator	Class::age = 2;	none

2	()
	()
	()
	[]
	->
	.
	++
	--
	const_cast
	dynamic_cast
	static_cast
	reinterpret_cast
	typeid
		Grouping operator
		Function call
		Member initialization
		Array access
		Member access from a pointer
		Member access from an object
		Post-increment
		Post-decrement
		Special cast
		Special cast
		Special cast
		Special cast
		Run-time type information
			(a + b) / 4;
			isdigit('1');
			c_tor(int x, int y) : _x(x), _y(y*10){};
			array[4] = 2;
			ptr->age = 34;
			obj.age = 34;
			for(i = 0; i < 10; i++) ...
			for(i = 10; i > 0; i--) ...
			const_cast<type_to>(type_from);
			dynamic_cast<type_to>(type_from);
			static_cast<type_to>(type_from);
			reinterpret_cast<type_to>(type_from);
			typeid(var).name();
			typeid(type).name();
				left to right

3	!
	~
	++
	--
	-
	+
	*
	&
	new
	new []
	delete
	delete []
	(type)
	sizeof
		Logical negation
		Bitwise complement
		Pre-increment
		Pre-decrement
		Unary minus
		Unary plus
		Dereference
		Address of
		Dynamic memory allocation
		Dynamic memory allocation of array
		Deallocate memory
		Deallocate memory of array
		Cast to a given type
		Return size in bytes
			if(!done) ...
			flags = ~flags;
			for(i = 0; i < 10; ++i) ...
			for(i = 10; i > 0; --i) ...
			int i = -1;
			int i = +1;
			data = *ptr;
			address = &obj;
			long *pVar = new long;
			long *array = new long[n];
			delete pVar;
			delete [] array;
			int i = (int) floatNum;
			int size = sizeof(floatNum);
				right to left

4	->* .*	Member pointer selector Member object selector	ptr->*var = 24; obj.*var = 24;	left to right

5	*
	/
	%
		Multiplication
		Division
		Modulus
			int i = 2 * 4;
			float f = 10 / 3;
			int rem = 4 % 3;
				left to right

<table>
<tr><td>6</td><td>+
-</td><td>Addition
Subtraction</td><td>int i = 2 + 3;
int i = 5 - 1;</td><td>left to right</td></tr>
<tr><td>7</td><td><<
>></td><td>Bitwise shift left
Bitwise shift right</td><td>int flags = 33 << 1;
int flags = 33 >> 1;</td><td>left to right</td></tr>
<tr><td>8</td><td><
<=
>
>=</td><td>Comparison less-than
Comparison less-than-or-equal-to
Comparison greater-than
Comparison geater-than-or-equal-to</td><td>if(i < 42) ...
if(i <= 42) ...
if(i > 42) ...
if(i >= 42) ...</td><td>left to right</td></tr>
<tr><td>9</td><td>==
!=</td><td>Comparison equal-to
Comparison not-equal-to</td><td>if(i == 42) ...
if(i != 42) ...</td><td>left to right</td></tr>
<tr><td>10</td><td>&</td><td>Bitwise AND</td><td>flags = flags & 42;</td><td>left to right</td></tr>
<tr><td>11</td><td>^</td><td>Bitwise exclusive OR</td><td>flags = flags ^ 42;</td><td>left to right</td></tr>
<tr><td>12</td><td>|</td><td>Bitwise inclusive (normal) OR</td><td>flags = flags | 42;</td><td>left to right</td></tr>
<tr><td>13</td><td>&&</td><td>Logical AND</td><td>if(conditionA && conditionB) ...</td><td>left to right</td></tr>
<tr><td>14</td><td>||</td><td>Logical OR</td><td>if(conditionA || conditionB) ...</td><td>left to right</td></tr>
<tr><td>15</td><td>? :</td><td>Ternary conditional (if-then-else)</td><td>int i = (a > b) ? a : b;</td><td>right to left</td></tr>
<tr><td>16</td><td>=
+=
-=
*=
/=
%=
&=
^=
|=
<<=
>>=</td><td>Assignment operator
Increment and assign
Decrement and assign
Multiply and assign
Divide and assign
Modulo and assign
Bitwise AND and assign
Bitwise exclusive OR and assign
Bitwise inclusive OR and assign
Bitwise shift left and assign
Bitwise shift right and assign</td><td>int a = b;
a += 3;
b -= 4;
a *= 5;
a /= 2;
a %= 3;
flags &= new_flags;
flags ^= new_flags;
flags |= new_flags;
flags <<= 2;
flags >>= 2;</td><td>right to left</td></tr>
<tr><td>17</td><td>throw</td><td>Throw exception</td><td>throw EClass("Message");</td><td>left to right</td></tr>
<tr><td>18</td><td>,</td><td>Sequential evaluation operator</td><td>for(i = 0, j = 0; i < 10; i++, j++) ...</td><td>left to right</td></tr>
</table>

One important aspect of C/C++ that is related to operator precedence is the **order of evaluation** and the **order of side effects** in expressions. In some circumstances, the order in which things happen is not defined. For example, consider the following code:

```
float x = 1;
x = x / ++x;
```

The value of *x* is not guaranteed to be consistent across different compilers, because it is not clear whether the computer should evaluate the left or the right side of the division first. Depending on which side is evaluated first, *x* could take a different value.
Furthermore, while ++x evaluates to x+1, the side effect of actually storing that new value in *x* could happen at different times, resulting in different values for *x*.

The bottom line is that expressions like the one above are horribly ambiguous and should be avoided at all costs. When in doubt, break a single ambiguous expression into multiple expressions to ensure that the order of evaluation is correct.

C/C++ Data Types

There are six data types for C: **void**, **_Bool**, **char**, **int**, **float**, and **double**.

Type	Description
`void`	associated with no data type
`_Bool`	boolean
`char`	character
`int`	integer
`float`	floating-point number
`double`	double precision floating-point number

C++ defines two more: **bool** and **wchar_t**.

Type	Description
`bool`	Boolean value, true or false
`wchar_t`	wide character

Type Modifiers

Several of these types can be modified using **signed**, **unsigned**, **short**, **long**, and **long long**. When one of these type modifiers is used by itself, a data type of **int** is assumed. A list of possible data types follows:

```
char
unsigned char
signed char
int
unsigned int
signed int
short int
unsigned short int
signed short int
long int
signed long int
unsigned long int
long long int
signed long long int
unsigned long long int
float
double
long double
```

Type Sizes and Ranges

The size and range of any data type is compiler and architecture dependent. The "cfloat" (or "float.h") header file often defines minimum and maximum values for the various data types. You can use the sizeof operator to determine the size of any data type, in bytes. However, many architectures implement data types of a standard size. **ints** and **floats** are often 32-bit, **chars** 8-bit, and **doubles** are usually 64-bit. **bools** are often implemented as 8-bit data types.

C Data Types (ARM)

The TASKING C compiler for the ARM architecture (**carm**) supports the following fundamental data types:

Type	C Type	Size (bit)	Align (bit)	Limits
Boolean	`_Bool`	8	8	0 or 1
Character	`char` `signed char`	8	8	-2^{7} .. $2^{7}-1$
	`unsigned char`	8	8	0 .. $2^{8}-1$
Integral	`short` `signed short`	16	16	-2^{15} .. $2^{15}-1$
	`unsigned short`	16	16	0 .. $2^{16}-1$
	`enum`	32	32	-2^{31} .. $2^{31}-1$

	`int`
	`signed int`
	`long`
	`signed long`
		32	32	-2^{31} .. $2^{31}-1$

	`unsigned int` `unsigned long`	32	32	0 .. $2^{32}-1$
	`long long` `signed long long`	64	32	-2^{63} .. $2^{63}-1$
	`unsigned long long`	64	64	0 .. $2^{64}-1$
Pointer	pointer to function or data	32	32	0 .. $2^{32}-1$
Floating-Point	`float`	32	32	-3.402E+38 .. -1.175E-38 1.175E-38 .. 3.402E+38
	`double` `long double`	64	64	-1.798E+308 .. -2.225E-308 2.225E-308 .. 1.798E+308

C Data Types (MicroBlaze)

The TASKING C compiler for the MicroBlaze architecture (**cmb**) supports the following fundamental data types:

Type	C Type	Size (bit)	Align (bit)	Limits
Boolean	`_Bool`	8	8	0 or 1
Character	`char` `signed char`	8	8	-2^7 .. 2^7-1
	`unsigned char`	8	8	0 .. 2^8-1
Integral	`short` `signed short`	16	16	-2^{15} .. $2^{15}-1$
	`unsigned short`	16	16	0 .. $2^{16}-1$
	`enum`	8 16 32	8 16 32	-2^7 .. 2^7-1 -2^{15} .. $2^{15}-1$ -2^{31} .. $2^{31}-1$

	`int`
	`signed int`
	`long`
	`signed long`
		32	32	-2^{31} .. $2^{31}-1$

	`unsigned int` `unsigned long`	32	32	0 .. $2^{32}-1$
	`long long` `signed long long`	64	32	-2^{63} .. $2^{63}-1$
	`unsigned long long`	64	32	0 .. $2^{64}-1$
Pointer	pointer to function or data	32	32	0 .. $2^{32}-1$
Floating-Point	`float`	32	32	-3.402E+38 .. -1.175E-38 1.175E-38 .. 3.402E+38
	`double` `long double`	64	32	-1.798E+308 .. -2.225E-308 2.225E-308 .. 1.798E+308

When you use the `enum` type, the compiler will use the smallest sufficient type (`char`, `short` or `int`), unless you use compiler option **`--integer-enumeration`** (always use integers for enumeration).

C Data Types (Nios II)

The TASKING C compiler for the Nios II architecture (**cnios**) supports the following fundamental data types:

Type	C Type	Size (bit)	Align (bit)	Limits
Boolean	`_Bool`	8	8	0 or 1
Character	`char` `signed char`	8	8	-2^7 .. 2^7-1
	`unsigned char`	8	8	0 .. 2^8-1
Integral	`short` `signed short`	16	16	-2^{15} .. $2^{15}-1$
	`unsigned short`	16	16	0 .. $2^{16}-1$
	`enum`	32	32	-2^{31} .. $2^{31}-1$

	`int`
	`signed int`
	`long`
	`signed long`
		32	32	-2^{31} .. $2^{31}-1$

	`unsigned int` `unsigned long`	32	32	0 .. $2^{32}-1$
	`long long` `signed long long`	64	32	-2^{63} .. $2^{63}-1$
	`unsigned long long`	64	32	0 .. $2^{64}-1$
Pointer	pointer to function or data	32	32	0 .. $2^{32}-1$
Floating-Point	`float`	32	32	-3.402E+38 .. -1.175E-38 1.175E-38 .. 3.402E+38
	`double` `long double`	64	32	-1.798E+308 .. -2.225E-308 2.225E-308 .. 1.798E+308

C Data Types (PowerPC)

The TASKING C compiler for the PowerPC architecture (**cppc**) supports the following fundamental data types:

Type	C Type	Size (bit)	Align (bit)	Limits
Boolean	`_Bool`	8	8	0 or 1
Character	`char` `signed char`	8	8	-2^{7} .. $2^{7}-1$
	`unsigned char`	8	8	0 .. $2^{8}-1$
Integral	`short` `signed short`	16	16	-2^{15} .. $2^{15}-1$
	`unsigned short`	16	16	0 .. $2^{16}-1$
	`enum`	32	32	-2^{31} .. $2^{31}-1$

	`int`
	`signed int`
	`long`
	`signed long`
		32	32	-2^{31} .. $2^{31}-1$

	`unsigned int` `unsigned long`	32	32	0 .. $2^{32}-1$
	`long long` `signed long long`	64	64	-2^{63} .. $2^{63}-1$
	`unsigned long long`	64	64	0 .. $2^{64}-1$
Pointer	pointer to function or data	32	32	0 .. $2^{32}-1$
Floating-Point	`float`	32	32	-3.402E+38 .. -1.175E-38 1.175E-38 .. 3.402E+38
	`double` `long double`	64	64	-1.798E+308 .. -2.225E-308 2.225E-308 .. 1.798E+308

C Data Types (TSK3000)

The TASKING C compiler for the TSK3000 architecture (**c3000**) supports the following fundamental data types:

Type	C Type	Size (bit)	Align (bit)	Limits
Boolean	`_Bool`	8	8	0 or 1
Character	`char` `signed char`	8	8	-2^{7} .. $2^{7}-1$
	`unsigned char`	8	8	0 .. $2^{8}-1$
Integral	`short` `signed short`	16	16	-2^{15} .. $2^{15}-1$
	`unsigned short`	16	16	0 .. $2^{16}-1$
	`enum`	32	32	-2^{31} .. $2^{31}-1$

	`int`
	`signed int`
	`long`
	`signed long`
		32	32	-2^{31} .. $2^{31}-1$

	`unsigned int` `unsigned long`	32	32	0 .. $2^{32}-1$
	`long long` `signed long long`	64	32	-2^{63} .. $2^{63}-1$
	`unsigned long long`	64	32	0 .. $2^{64}-1$
Pointer	pointer to function or data	32	32	0 .. $2^{32}-1$
Floating-Point	`float`	32	32	-3.402E+38 .. -1.175E-38 1.175E-38 .. 3.402E+38
	`double` `long double`	64	32	-1.798E+308 .. -2.225E-308 2.225E-308 .. 1.798E+308

C Data Types (TSK51x/TSK52x)

The TASKING C compiler for the TSK51x/TSK52x architecture (**c51**) supports the following data types:

Type	C Type	Size (bit)	Align (bit)	Limits
Bit	`__bit`	1	1	0 or 1
Boolean	`_Bool`	1	8	0 or 1
Character	`char` `signed char`	8	8	-2^7 .. 2^7-1
	`unsigned char`	8	8	0 .. 2^8-1

Integral	`short`
	`signed short`
	`int`
	`signed int`
		16	8	-2^{15} .. $2^{15}-1$

	`enum`	1 8 16	1 8 8	0 or 1 -2^7 .. 2^7-1 -2^{15} .. $2^{15}-1$
	`unsigned short` `unsigned int`	16	8	0 .. $2^{16}-1$
	`long` `signed long`	32	8	-2^{31} .. $2^{31}-1$
	`unsigned long`	32	8	0 .. $2^{32}-1$
	`long long` `signed long long`	32	8	-2^{31} .. $2^{31}-1$
	`unsigned long long`	32	8	0 .. $2^{32}-1$
Pointer	pointer to __sfr, __bsfr, __data, __bdata, __idata, __pdata or __bit	8	8	0 .. 2^8-1
	pointer to function, __xdata or __rom	16	8	0 .. $2^{16}-1$
Floating-Point	`float`	32	8	-3.402E+38 .. -1.175E-38 1.175E-38 .. 3.402E+38
	`double` `long double`	32	8	-3.402E+38 .. -1.175E-38 1.175E-38 .. 3.402E+38

The `double` and `long double` types are always treated as `float`.
When you use the `enum` type, the compiler will use the smallest sufficient type (`__bit`, `char`, `int`), unless you use compiler option **`--integer-enumeration`** (always use 16-bit integers for enumeration).

C Data Types (TSK80x)

The TASKING C compiler for the TSK80x architecture (**cz80**) supports the following fundamental data types:

Type	C Type	Size (bit)	Align (bit)	Limits
Boolean	`_Bool`	1	8	0 or 1
Character	`char` `signed char`	8	8	-2^{7} .. 2^{7}-1
	`unsigned char`	8	8	0 .. 2^{8}-1

Integral	`short`
	`signed short`
	`int`
	`signed int`
		16	8	-2^{15} .. 2^{15}-1

	`enum`	8 16	8 8	-2^{7} .. 2^{7}-1 -2^{15} .. 2^{15}-1
	`unsigned short` `unsigned int`	16	8	0 .. 2^{16}-1
	`long` `signed long`	32	8	-2^{31} .. 2^{31}-1
	`unsigned long`	32	8	0 .. 2^{32}-1
	`long long` `signed long long`	32	8	-2^{31} .. 2^{31}-1
	`unsigned long long`	32	8	0 .. 2^{32}-1
Pointer	pointer to __sfr8	8	8	0 .. 2^{8}-1
	pointer __sfr, data or function	16	8	0 .. 2^{16}-1
Floating-Point	`float`	32	8	-3.402E+38 .. -1.175E-38 1.175E-38 .. 3.402E+38
	`double` `long double`	32	8	-3.402E+38 .. -1.175E-38 1.175E-38 .. 3.402E+38

The `long long` types are treated as `long`.
The `double` and `long double` types are always treated as `float`.
When you use the `enum` type, the compiler will use the smallest sufficient type (`char` or `int`), unless you use compiler option **`--integer-enumeration`** (always use 16-bit integers for enumeration).

Memory Types

Depending on the target for which you are writing C source code, several memories or memory types may be available for placing data objects. Memory types can either be physically different memories or can be defined ranges in a single memory.

If more than one memory type is available for the target, you can specify in which (part of) the memory a variable must be placed. You can do this with *`memory type qualifiers`*. Depending on the target and its available memory types, several memory type qualifiers are supported.

Memory Types (MicroBlaze)

Qualifier	Description
`__no_sdata`	Direct addressable RAM
`__sdata`	Direct short (near) addressable RAM (Small data, 64k)
`__sfr`	(For compatibilty with special function registers)
`__rom`	Data defined with this qualifier is placed in ROM. This section is excluded from automatic initialization by the startup code. `__rom` always implies the type qualifier `const`.

By default, all data objects smaller than 4 bytes are placed in small data (sdata) sections. With the `__no_sdata` and `__sdata` keywords, you can overrule this default and either force larger data objects in sdata or prevent smaller data objects from being placed in sdata.

Example

```
__rom  char  text[] = "No smoking";
long long l = 1234;                  // long long reserved in data (by default)
__sdata  long long  k = 1234;        // long long reserved in sdata
```

The memory type qualifiers are treated like any other data type specifier (such as `unsigned`). This means the examples above can also be declared as:

```
char __rom  text[] = "No smoking";
long long __sdata k = 1234;
```

The `__sfr` keyword lets you define a variable as a "special function register". Though special function registers are not available for the MicroBlaze, the compiler accepts the `__sfr` keyword as a qualifier for compatibility reasons. Variables declared with `__sfr` have some special characteristics.

Because special function registers are dealing with I/O, it is incorrect to optimize away the access to them. Therefore, the compiler deals with `__sfr` variables as if they were declared with the `volatile` qualifier.

Non-initialized global `__sfr` variables are not cleared at startup. For example:

```
__sfr int i;                          // global __sfr variable not cleared
```

It is not allowed to initialize global `__sfr` variables and they are not initialized at startup. For example:

```
__sfr int j=10;                      // not allowed to initialize global __sfr variable
```

Memory Types (Nios II)

Qualifier	Description
`__no_sdata`	Direct addressable RAM
`__sdata`	Direct short addressable RAM (Small data, +/- 32kB offset from global pointer register $gp)

By default, all data objects smaller than 4 bytes are placed in small data (sdata) sections. With the `__no_sdata` and `__sdata` keywords, you can overrule this default and either force larger data objects in sdata or prevent smaller data objects from being placed in sdata.

Example

```
long long l = 1234;                  // long long reserved in data (by default)

__sdata  long long  k = 1234;        // long long reserved in sdata
```

The memory type qualifiers are treated like any other data type specifier (such as `unsigned`). This means the examples above can also be declared as:

```
long long __sdata k = 1234;
```

Memory Types (PowerPC)

Qualifier	Description
`__no_sdata`	Direct addressable RAM
`__sdata`	Direct short addressable RAM (Small data, +/- 32kB offset from global pointer register $gp)

The PowerPC in fact has two small data memories, sdata and sdata2, both with a size of 64kB. By default, all data objects smaller than 4 bytes are placed in sdata or sdata2 (non-constant data is placed in sdata whereas constant data is placed in sdata2). With the `__no_sdata` and `__sdata` keywords, you overrule this default.

Example

```
long long l = 1234;                // long long reserved in data (by default)
const long long k = 1234;          // long long reserved in rodata (by default)

__sdata  long long  m;             // long long reserved in sdata
const __sdata  long long n = 1234; // long long in sdata2
```

The memory type qualifiers are treated like any other data type specifier (such as `unsigned`). This means the example above can also be declared as:

```
long long  __sdata  m = 1234;
const long long __sdata  n = 1234;
```

Memory Types (TSK3000)

Qualifier	Description
`__no_sdata`	Direct addressable RAM
`__sdata`	Direct short addressable RAM (Small data, +/- 32kB offset from global pointer register $gp)

By default, all data objects smaller than 4 bytes are placed in small data (sdata) sections. With the `__no_sdata` and `__sdata` keywords, you can overrule this default and either force larger data objects in sdata or prevent smaller data objects from being placed in sdata.

Example

```
long long l = 1234;                  // long long reserved in data (by default)

__sdata  long long  k = 1234;        // long long reserved in sdata
```

The memory type qualifiers are treated like any other data type specifier (such as `unsigned`). This means the examples above can also be declared as:

```
long long __sdata k = 1234;
```

Memory Types (TSK51x/TSK52x)

Qualifier	Description
`__data`	Direct addressable on-chip RAM
`__sfr`	Defines a special function register. Special optimizations are performed on this type of variables.
`__bsfr`	Bit-addressable special function register
`__idata`	Indirect addressable on-chip RAM
`__bdata`	Bit-addressable on-chip RAM
`__xdata`	External RAM
`__pdata`	One 256 bytes page within external RAM
`__rom`	Data defined with this qualifier is placed in ROM. This section is excluded from automatic initialization by the startup code. `__rom` always implies the type qualifier `const`.

If you do not specify a memory type qualifier for the TSK51x/TSK52x, the memory type for the variable depends on the default of the selected memory model (project options).

Memory Model	Description	Max RAM size	Default memory type
small	direct addressable internal RAM	128 bytes	`__data`
auxiliary page	one page of external RAM	256 bytes	`__pdata`
large	external RAM	64 kB	`__xdata`

Example

```
__data  char  c;
__rom    char  text[] = "No smoking";
__xdata  int  array[10][4];
__idata  long  l;
```

The memory type qualifiers are treated like any other data type specifier (such as `unsigned`). This means the example above can also be declared as:

```
char __data  c;
char __rom    text[] = "No smoking";
int  __xdata  array[10][4];
long __idata  l;
```

Memory Types (TSK80x)

Qualifier	Description
`__sfr`	Defines a special function register for access of peripherals via the TSK80x I/O space. Special optimizations are performed on this type of variables.
`__sfr8`	Defines an 8-bit special function register for access of peripherals via the 8-bit addressable part of the TSK80x I/O space.
`__rom`	Data defined with this qualifier is placed in ROM. This section is excluded from automatic initialization by the startup code. `__rom` always implies the type qualifier `const`.

Example

```
__rom  char  text[] = "No smoking";
```

The memory type qualifiers are treated like any other data type specifier (such as `unsigned`). This means the example above can also be declared as:

```
char  __rom  text[] = "No smoking";
```

Complexity

There are different measurements of the speed of any given algorithm. Given an input size of **N**, they can be described as follows:

Name	**Speed**	**Description**	**Formula**
exponential time	slow	takes an amount of time proportional to a constant raised to the **N**th power	K^**N**
polynomial time	fast	takes an amount of time proportional to **N** raised to some constant power	**N**^K
linear time	faster	takes an amount of time directly proportional to **N**	K * **N**
logarithmic time	much faster	takes an amount of time proportional to the logarithm of **N**	K * log(**N**)
constant time	fastest	takes a fixed amount of time, no matter how large the input is	K

Constant Escape Sequences

The following escape sequences can be used to define certain special characters within strings:

Escape Sequence	Description
\'	Single quote
\"	Double quote
\\	Backslash
\nnn	Octal number (nnn)
\0	Null character (really just the octal number zero)
\a	Audible bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\xnnn	Hexadecimal number (nnn)

An example of this is contained in the following code:

```
printf( "This\nis\na\ntest\n\nShe said, \"How are you?\"\n" );
```

which would display

```
This
is
a
test

She said, "How are you?"
```

ASCII Chart

The following chart contains ASCII decimal, octal, hexadecimal and character codes for values from 0 to 127.

Decimal	Octal	Hex	Character	Description
0	0	00	NUL	
1	1	01	SOH	start of header
2	2	02	STX	start of text
3	3	03	ETX	end of text
4	4	04	EOT	end of transmission
5	5	05	ENQ	enquiry
6	6	06	ACK	acknowledge
7	7	07	BEL	bell
8	10	08	BS	backspace
9	11	09	HT	horizontal tab
10	12	0A	LF	line feed
11	13	0B	VT	vertical tab
12	14	0C	FF	form feed
13	15	0D	CR	carriage return
14	16	0E	SO	shift out
15	17	0F	SI	shift in
16	20	10	DLE	data link escape
17	21	11	DC1	no assignment, but usually XON
18	22	12	DC2	
19	23	13	DC3	no assignment, but usually XOFF
20	24	14	DC4	
21	25	15	NAK	negative acknowledge
22	26	16	SYN	synchronous idle
23	27	17	ETB	end of transmission block
24	30	18	CAN	cancel
25	31	19	EM	end of medium
26	32	1A	SUB	substitute
27	33	1B	ESC	escape
28	34	1C	FS	file seperator
29	35	1D	GS	group seperator
30	36	1E	RS	record seperator
31	37	1F	US	unit seperator
32	40	20	SPC	space

Decimal	Octal	Hex	Character	Description
33	41	21	!	
34	42	22	"	
35	43	23	#	
36	44	24	$	
37	45	25	%	
38	46	26	&	
39	47	27	'	
40	50	28	(	
41	51	29	)	
42	52	2A	*	
43	53	2B	+	
44	54	2C	,	
45	55	2D	-	
46	56	2E	.	
47	57	2F	/	
48	60	30	0	
49	61	31	1	
50	62	32	2	
51	63	33	3	
52	64	34	4	
53	65	35	5	
54	66	36	6	
55	67	37	7	
56	70	38	8	
57	71	39	9	
58	72	3A	:	
59	73	3B	;	
60	74	3C	<	
61	75	3D	=	
62	76	3E	>	
63	77	3F	?	
64	100	40	@	
65	101	41	A	
66	102	42	B	
67	103	43	C	
68	104	44	D	
69	105	45	E	

Decimal	Octal	Hex	Character	Description
70	106	46	F	
71	107	47	G	
72	110	48	H	
73	111	49	I	
74	112	4A	J	
75	113	4B	K	
76	114	4C	L	
77	115	4D	M	
78	116	4E	N	
79	117	4F	O	
80	120	50	P	
81	121	51	Q	
82	122	52	R	
83	123	53	S	
84	124	54	T	
85	125	55	U	
86	126	56	V	
87	127	57	W	
88	130	58	X	
89	131	59	Y	
90	132	5A	Z	
91	133	5B	[	
92	134	5C	\	
93	135	5D	]	
94	136	5E	^	
95	137	5F	_	
96	140	60	`	
97	141	61	a	
98	142	62	b	
99	143	63	c	
100	144	64	d	
101	145	65	e	
102	146	66	f	
103	147	67	g	
104	150	68	h	
105	151	69	i	
106	152	6A	j	

Decimal	Octal	Hex	Character	Description
107	153	6B	k	
108	154	6C	l	
109	155	6D	m	
110	156	6E	n	
111	157	6F	o	
112	160	70	p	
113	161	71	q	
114	162	72	r	
115	163	73	s	
116	164	74	t	
117	165	75	u	
118	166	76	v	
119	167	77	w	
120	170	78	x	
121	171	79	y	
122	172	7A	z	
123	173	7B	{	
124	174	7C	|	
125	175	7D	}	
126	176	7E	~	
127	177	7F	DEL	delete

Pre-processor Commands

The following is a list of all pre-processor commands in the standard C language.

#, ##	manipulate strings
#define	define variables
#error	display an error message
#if, #ifdef, #ifndef, #else, #elif, #endif	conditional operators
#include	insert the contents of another file
#line	set line and file information
#pragma	implementation specific command
#undef	used to undefine variables
Predefined preprocessor variables	miscellaneous preprocessor variables

Pre-processor command: #,

The # and ## operators are used with the #define macro. Using # causes the first argument after the # to be returned as a string in quotes. Using ## concatenates what's before the ## with what's after it.

Example

For example, the command

```
#define to_string( s ) # s
```

will make the compiler turn this command

```
printf(to_string( Hello World! ));
```

into

```
printf("Hello World!");
```

Here is an example of the ## command:

```
#define concatenate( x, y ) x ## y
```

This code will make the compiler turn

```
int concatenate( x, y ) = 10;
```

into

```
int xy = 10;
```

which will, of course, assign 10 to the integer variable 'xy'.

Pre-processor command: #define

Syntax

```
#define macro-name replacement-string
```

The #define command is used to make substitutions throughout the file in which it is located. In other words, #define causes the compiler to go through the file, replacing every occurrence of *macro-name* with *replacement-string*. The replacement string stops at the end of the line.

Example

Here's a typical use for a #define (at least in C):

```
#define TRUE 1
#define FALSE 0
...
int done = 0;
while( done != TRUE )
{
    ...
}
```

Another feature of the #define command is that it can take arguments, making it rather useful as a pseudo-function creator. Consider the following code:

```
#define absolute_value( x ) ( ((x) < 0) ? -(x) : (x) )
...
int num = -1;
while( absolute_value( num ) )
{
    ...
}
```

It's generally a good idea to use extra parentheses when using complex macros. Notice that in the above example, the variable "x" is always within it's own set of parentheses. This way, it will be evaluated in whole, before being compared to 0 or multiplied by -1. Also, the entire macro is surrounded by parentheses, to prevent it from being contaminated by other code. If you're not careful, you run the risk of having the compiler misinterpret your code.

Here is an example of how to use the #define command to create a general purpose incrementing for loop that prints out the integers 1 through 20:

```
#define count_up( v, low, high ) \
  for( (v) = (low); (v) <= (high); (v)++ )

...

int i;
count_up( i, 1, 20 )
{
  printf( "i is %d\n", i );
}
```

Pre-processor command: #error

Syntax

```
#error message
```

The #error command simply causes the compiler to stop when it is encountered. When an #error is encountered, the compiler spits out the line number and whatever *message* is. This command is mostly used for debugging.

Pre-processor command: #if, #ifdef, #ifndef, #else, #elif, #endif

These commands give simple logic control to the compiler. As a file is being compiled, you can use these commands to cause certain lines of code to be included or not included.

```
#if expression
```

If the value of expression is true, then the code that immediately follows the command will be compiled.

```
#ifdef macro
```

If the *macro* has been defined by a #define statement, then the code immediately following the command will be compiled.

```
#ifndef macro
```

If the *macro* has not been defined by a #define statement, then the code immediately following the command will be compiled.

A few side notes: The command #elif is simply a horribly truncated way to say "elseif" and works like you think it would. You can also throw in a "defined" or "!defined" after an #if to get added functionality.

Example

Here's an example of all these:

```
#ifdef DEBUG
  printf("This is the test version, i=%d\n", i);
#else
  printf("This is the production version!\n");
#endif
```

Notice how that second example enables you to compile the same C source either to a debug version or to a production version.

Pre-processor command: #include

Syntax

```
#include <filename>
#include "filename"
```

This command slurps in a file and inserts it at the current location. The main difference between the syntax of the two items is that if *`filename`* is enclosed in angled brackets, then the compiler searches for it somehow. If it is enclosed in quotes, then the compiler doesn't search very hard for the file.

While the behavior of these two searches is up to the compiler, usually the angled brackets means to search through the standard library directories, while the quotes indicate a search in the current directory. For standard libraries, the #include commands don't need to map directly to filenames. It is possible to use *standard headers* instead:

```
#include <iostream>
```

Pre-processor command: #line

Syntax

```
#line line_number "filename"
```

The #line command is simply used to change the value of the __LINE__ and __FILE__ variables. The filename is optional. The __LINE__ and __FILE__ variables represent the current file and which line is being read. The command

```
#line 10 "main.cpp"
```

changes the current line number to 10, and the current file to "main.cpp".

Pre-processor command: #pragma

The #pragma command gives the programmer the ability to tell the compiler to do certain things. Since the #pragma command is implementation specific, uses vary from compiler to compiler. One option might be to trace program execution.

Pre-processor command: #undef

The #undef command undefines a previously defined macro variable, such as a variable defined by a #define.

Predefined preprocessor variables

Syntax

```
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
__STDC__
```

The following variables can vary by compiler, but generally work:

- The __LINE__ and __FILE__ variables represent the current line and current file being processed.
- The __DATE__ variable contains the current date, in the form month/day/year. This is the date that the file was compiled, not necessarily the current date.
- The __TIME__ variable represents the current time, in the form hour:minute:second. This is the time that the file was compiled, not necessarily the current time.
- The __cplusplus variable is only defined when compiling a C++ program. In some older compilers, this is also called c_plusplus.
- The __STDC__ variable is defined when compiling a C program, and may also be defined when compiling C++.

C/C++ Keywords

The following is a list of all keywords that exist in the standard C language.

C/C++ Keywords

asm	insert an assembly instruction
auto	declare a local variable
bool	declare a boolean variable
break	break out of a loop
case	a block of code in a switch statement
catch	handles exceptions from throw
char	declare a character variable
class	declare a class
const	declare immutable data or functions that do not change data
const_cast	cast from const variables
continue	bypass iterations of a loop
default	default handler in a case statement
delete	make memory available
do	looping construct
double	declare a double precision floating-point variable
dynamic_cast	perform run-time casts
else	alternate case for an if statement
enum	create enumeration types
explicit	only use constructors when they exactly match
export	allows template definitions to be separated from their declarations
extern	tell the compiler about variables defined elsewhere
false	the boolean value of false
float	declare a floating-point variable
for	looping construct
friend	grant non-member function access to private data
goto	jump to a different part of the program
if	execute code based off of the result of a test
inline	optimize calls to short functions
int	declare a integer variable
long	declare a long integer variable
mutable	override a const variable
namespace	partition the global namespace by defining a scope
new	allocate dynamic memory for a new variable
operator	create overloaded operator functions
private	declare private members of a class

protected	declare protected members of a class
public	declare public members of a class
register	request that a variable be optimized for speed
reinterpret_cast	change the type of a variable
restrict	inform compiler about access restrictions for optimizations
return	return from a function
short	declare a short integer variable
signed	modify variable type declarations
sizeof	return the size of a variable or type
static	create permanent storage for a variable
static_cast	perform a nonpolymorphic cast
struct	define a new structure
switch	execute code based off of different possible values for a variable
template	create generic functions
this	a pointer to the current object
throw	throws an exception
true	the boolean value of true
try	execute code that can throw an exception
typedef	create a new type name from an existing type
typeid	describes an object
typename	declare a class or undefined type
union	a structure that assigns multiple variables to the same memory location
unsigned	declare an unsigned integer variable
using	import complete or partial namespaces into the current scope
virtual	create a function that can be overridden by a derived class
void	declare functions or data with no associated data type
volatile	warn the compiler about variables that can be modified unexpectedly
wchar_t	declare a wide-character variable
while	looping construct

C/C++ keyword: asm

Syntax

```
asm( "instruction" );
```

The `asm` command allows you to insert assembly language commands directly into your code. The `__asm__` keyword is recognized and is equivalent to the `asm` token. Extended syntax is supported to indicate how assembly operands map to C/C++ variables.

Example

```
asm("fsinx %1,%0" : "=f"(x) : "f"(a));
  // Map the output operand on "x",
  // and the input operand on "a".
```

C/C++ keyword: auto

The keyword auto is used to declare local variables with automatic (i.e. not static) storage duration.

The auto keyword is purely optional and is rarely used.

C/C++ keyword: bool

The keyword bool is used to declare Boolean logic variables; that is, variables which can be either true or false.

For example, the following code declares a boolean variable called *done*, initializes it to false, and then loops until that variable is set to true.

```
bool done = false;
while( !done )
{
...
}
```

Also see the C/C++ data types.

C/C++ keyword: break

The break keyword is used to break out of a do, for, or while loop. It is also used to finish each clause of a switch statement, keeping the program from "falling through" to the next case in the code. An example:

```
while( x < 100 )
{
  if( x < 0 )
    break;
  printf("%d\n", x);
  x++;
}
```

A given break statement will break out of only the closest loop, no further. If you have a triply-nested for loop, for example, you might want to include extra logic or a goto statement to break out of the loop.

C/C++ keyword: case

The case keyword is used to test a variable against a certain value in a switch statement.

C/C++ keyword: catch

The catch statement handles exceptions generated by the throw statement.

C/C++ keyword: char

The char keyword is used to declare character variables. For more information about variable types, see the C/C++ data types.

C/C++ keyword: class

Syntax

```
class class-name : inheritance-list
{
  private-members-list;
protected:
  protected-members-list;
public:
  public-members-list;
} object-list;
```

The class keyword allows you to create new classes. *class-name* is the name of the class that you wish to create, and *inheritance-list* is an optional list of classes inherited by the new class. Members of the class are private by default, unless listed under either the protected or public labels. *object-list* can be used to immediately instantiate one or more instances of the class, and is also optional.

Example

```
class Date
{
  int Day;
  int Month;
  int Year;
public:
  void display();
};
```

C/C++ keyword: const

The const keyword can be used to tell the compiler that a certain variable should not be modified once it has been initialized. It can also be used to declare functions of a class that do not alter any class data.

C/C++ keyword: const_cast

Syntax

```
TYPE const_cast<TYPE> (object);
```

The const_cast keyword can be used to remove the **const** or **volatile** property from an object. The target data type must be the same as the source type, except (of course) that the target type doesn't have to have the same const qualifier. The type TYPE must be a pointer or reference type.

For example, the following code uses const_cast to remove the const qualifier from an object:

```
class Foo
{
public:
  void func() {} // a non-const member function
};

void someFunction( const Foo& f )
{
  f.func();      // compile error: cannot call a non-const
                 // function on a const reference
  Foo &fRef = const_cast<Foo&>(f);
  fRef.func();  // okay
}
```

C/C++ keyword: continue

The continue statement can be used to bypass iterations of a given loop.

For example, the following code will display all of the numbers between 0 and 20 except 10:

```
for( int i = 0; i < 21; i++ )
{
  if( i == 10 )
  {
    continue;
  }
  printf("%d ", i);
}
```

C/C++ keyword: default

A default case in the switch statement.

C/C++ keyword: delete

Syntax

```
delete p;
delete[] pArray;
```

The delete operator frees the memory pointed to by *p*. The argument should have been previously allocated by a call to new or 0. The second form of delete should be used to delete an array that was allocated with "new []". If (in either forms) the argument is 0 (NULL), nothing is done.

C/C++ keyword: do

Syntax

```
do
{
statement-list;
} while( condition );
```

The do construct evaluates the given `statement-list` repeatedly, until `condition` becomes false. Note that every do loop will evaluate its statement list at least once, because the terminating condition is tested at the end of the loop.

C/C++ keyword: double

The double keyword is used to declare double precision floating-point variables. Also see the C/C++ data types.

C/C++ keyword: dynamic_cast

Syntax

```
TYPE& dynamic_cast<TYPE&> (object);
TYPE* dynamic_cast<TYPE*> (object);
```

The dynamic_cast keyword casts a datum from one type to another, performing a run-time check to ensure the validity of the cast.

If you attempt to cast to a pointer type, and that type is not an actual type of the argument object, then the result of the cast will be **NULL**.

If you attempt to cast to a reference type, and that type is not an actual type of the argument object, then the cast will throw a **std::bad_cast** exception.

```
struct A
{
  virtual void f() { }
};
struct B : public A { };
struct C { };

void f ()
{
  A a;
  B b;

  A* ap = &b
  B* b1 = dynamic_cast<B*> (&a);  // NULL, because 'a' is not a 'B'
  B* b2 = dynamic_cast<B*> (ap);  // 'b'
  C* c = dynamic_cast<C*> (ap);  // NULL

  A& ar = dynamic_cast<A&> (*ap); // OK
  B& br = dynamic_cast<B&> (*ap); // OK
  C& cr = dynamic_cast<C&> (*ap); // std::bad_cast
}
```

C/C++ keyword: else

The else keyword is used as an alternative case for the if statement.

C/C++ keyword: enum

Syntax

```
enum name {name-list} var-list;
```

The enum keyword is used to create an enumerated type named *name* that consists of the elements in *name-list*. The *var-list* argument is optional, and can be used to create instances of the type along with the declaration. For example, the following code creates an enumerated type for colors:

```
enum ColorT {red, orange, yellow, green, blue, indigo, violet};
...
enum ColorT c1 = indigo;  // see note
if( c1 == indigo )
{
  printf("c1 is indigo\n");
}
```

In the above example, the effect of the enumeration is to introduce several new constants named *red*, *orange*, *yellow*, etc. By default, these constants are assigned consecutive integer values starting at zero. You can change the values of those constants, as shown by the next example:

```
enum ColorT { red = 10, blue = 15, green };
...
enum ColorT c = green;  // see note
printf("c is %d\n", c);
```

When executed, the above code will display the following output:

```
c is 16
```

Note: in C++ you can omit the enum keyword whenever you create an instance of an enumerated type.

C/C++ keyword: explicit

When a constructor is specified as explicit, no automatic conversion will be used with that constructor -- but parameters passed to the constructor may still be converted. For example:

```
struct foo
{
  explicit foo( int a )
    : a_( a )
  { }

  int a_;
};

int bar( const foo & f )
{
  return f.a_;
}
```

```
bar( 1 );  // fails because an implicit conversion from int to foo
           // is forbidden by explicit.

bar( foo( 1 ) );  // works -- explicit call to explicit constructor.

bar( static_cast<foo>( 1 ) );  // works -- call to explicit constructor via explicit cast.

bar( foo( 1.0 ) );  // works -- explicit call to explicit constructor
                    // with automatic conversion from float to int.
```

C/C++ keyword: export

The `export` keyword is used to allow definitions of C++ templates to be separated from their declarations.

Exporting a class template is equivalent to exporting each of its static data members and each of its non-inline member functions. An exported template is special because its definition does not need to be present in a translation unit that uses that template. In other words, the definition of an exported (non-class) template does not need to be explicitly or implicitly included in a translation unit that instantiates that template. For example, the following is a valid C++ program consisting of two separate translation units:

```
// File 1:
#include <stdio.h>
static void trace() { printf("File 1\n"); }

export template<class T> T const& min(T const&, T const&);
int main()
{
  trace();
  return min(2, 3);
}

// File 2:
#include <stdio.h>
static void trace() { printf("File 2\n"); }

export template<class T> T const& min(T const &a, T const &b)
{
  trace();
  return a<b? a: b;
}
```

Note that these two files are separate translation units: one is not included in the other. That allows the two functions `trace()` to coexist (with internal linkage).

C/C++ keyword: extern

The extern keyword is used to inform the compiler about variables declared outside of the current scope. Variables described by extern statements will not have any space allocated for them, as they should be properly defined elsewhere.

Extern statements are frequently used to allow data to span the scope of multiple files.

When applied to function declarations, the additional "C" or "C++" string literal will change name mangling when compiling under the opposite language. That is,

```
extern "C" int plain_c_func(int param);
```

allows C++ code to execute a C library function `plain_c_func`.

C/C++ keyword: false

The Boolean value of "false".

C/C++ keyword: float

The float keyword is used to declare floating-point variables. Also see the C/C++ data types.

C/C++ keyword: for

Syntax

```
for( initialization; test-condition; increment )
{
  statement-list;
}
```

The for construct is a general looping mechanism consisting of 4 parts:

1. the *`initialization`*, which consists of 0 or more comma-delimited variable initialization statements
2. the *`test-condition`*, which is evaluated to determine if the execution of the for loop will continue
3. the *`increment`*, which consists of 0 or more comma-delimited statements that increment variables
4. and the *`statement-list`*, which consists of 0 or more statements that will be executed each time the loop is executed.

For example:

```
for( int i = 0; i < 10; i++ )
{
  printf("i is %d", i);
}
int j, k;
for( j = 0, k = 10;
      j < k;
      j++, k-- )
{
  printf("j is %d and k is %d\n", j, k);
}
for( ; ; )
{
  // loop forever!
}
```

C/C++ keyword: friend

The friend keyword allows classes or functions not normally associated with a given class to have access to the private data of that class.

C/C++ keyword: goto

Syntax

```
goto labelA;
...
labelA:
```

The goto statement causes the current thread of execution to jump to the specified label. While the use of the goto statement is generally considered harmful, it can occasionally be useful. For example, it may be cleaner to use a goto to break out of a deeply-nested for loop, compared to the space and time that extra break logic would consume.

C/C++ keyword: if

Syntax

```
if( conditionA )
{
  statement-listA;
}
else if( conditionB )
{
  statement-listB;
}
...

else
{
  statement-listN;
}
```

The if construct is a branching mechanism that allows different code to execute under different conditions. The conditions are evaluated in order, and the statement-list of the first condition to evaluate to true is executed. If no conditions evaluate to true and an else statement is present, then the statement list within the else block will be executed. All of the else blocks are optional.

C keyword: inline

Syntax

```
inline int functionA( int i )
{
    // inline this function
}
```

The inline keyword requests that the compiler expand a given function in place, as opposed to inserting a call to that function. The inline keyword is a request, not a command, and the compiler is free to ignore it for whatever reason.

With the inline keyword you ask the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself.

Example

```
inline unsigned int abs(int val)
{
  unsigned int abs_val = val;
  if (val < 0) abs_val = -val;
  return abs_val;
}
```

You must define inline functions in the same source module as in which you call the function, because the compiler only inlines a function in the module that contains the function definition. When you need to call the inline function from several source modules, you must include the definition of the inline function in each module (for example using a header file).

When a function declaration is included in a class definition, the compiler should try to automatically inline that function. No inline keyword is necessary in this case.

C/C++ keyword: int

The int keyword is used to declare integer variables. Also see the C/C++ data types.

C/C++ keyword: long

The long keyword is a data type modifier that is used to declare long integer variables. For more information on long, see the C/C++ data types.

C/C++ keyword: mutable

The mutable keyword overrides any enclosing const statement. A mutable member of a const object can be modified.

C/C++ keyword: namespace

Syntax

```
namespace name
{
  declaration-list;
}
```

The namespace keyword allows you to create a new scope. The name is optional, and can be omitted to create an unnamed namespace. Once you create a namespace, you'll have to refer to it explicitly or use the using keyword.

Example

```
namespace CartoonNameSpace
{
  int HomersAge;
  void incrementHomersAge()
  {
    HomersAge++;
  }
}
int main()
{
  ...
  CartoonNameSpace::HomersAge = 39;
  CartoonNameSpace::incrementHomersAge();
  printf("%d\n", CartoonNameSpace::HomersAge);
  ...
}
```

C/C++ keyword: new

Syntax

```
pointer = new type;
pointer = new type( initializer );
pointer = new type[size];
pointer = new( arg-list ) type...
```

The new operator (valid only in C++) allocates a new chunk of memory to hold a variable of type *type* and returns a pointer to that memory. An optional initializer can be used to initialize the memory (or, when type is a class, to provide arguments to the constructor).

Allocating arrays can be accomplished by providing a *size* parameter in brackets (note that in this case no initializer can be given, so the type must be default-constructible).

The optional *arg-list* parameter can be used with any of the other formats to pass a variable number of arguments to an overloaded version of new(). For example, the following code shows how the new() function can be overloaded for a class and then passed arbitrary arguments:

```
class Base
{
public:
  Base() { }

  void *operator new( unsigned int size, string str )
  {
    printf("Logging an allocation of %d bytes for new object '%s'\n", size, str);
    return malloc( size );
  }

  int var;
  double var2;
};

...

Base* b = new ("Base instance 1") Base;
```

If an int is 4 bytes and a double is 8 bytes, the above code generates the following output when run:

```
Logging an allocation of 12 bytes for new object 'Base instance 1'
```

C/C++ keyword: operator

Syntax

```
return-type class-name::operator#(parameter-list)
{
...
}
return-type operator#(parameter-list)
{
...
}
```

The operator keyword is used to overload operators. The sharp sign (#) listed above in the syntax description represents the operator which will be overloaded. If part of a class, the *class-name* should be specified. For unary operators, *parameter-list* should be empty, and for binary operators, *parameter-list* should contain the operand on the right side of the operator (the operand on the left side is passed as this).

For the non-member operator overload function, the operand on the left side should be passed as the first parameter and the operand on the right side should be passed as the second parameter.

You cannot overload the #, ##, ., :, .*, or ? tokens.

C/C++ keyword: private

Private data of a class can only be accessed by members of that class, except when friend is used. The private keyword can also be used to inherit a base class privately, which causes all public and protected members of the base class to become private members of the derived class.

C/C++ keyword: protected

Protected data are private to their own class but can be inherited by derived classes. The protected keyword can also be used as an inheritance specifier, which causes all public and protected members of the base class to become protected members of the derived class.

C/C++ keyword: public

Public data in a class are accessible to everyone. The public keyword can also be used as an inheritance specifier, which causes all public and protected members of the base class to become public and protected members of the derived class.

C/C++ keyword: register

The register keyword requests that a variable be optimized for speed, and fell out of common use when computers became better at most code optimizations than humans.

C/C++ keyword: reinterpret_cast

Syntax

```
TYPE reinterpret_cast<TYPE> (object);
```

The reinterpret_cast operator changes one data type into another. It should be used to cast between incompatible pointer types.

C/C++ keyword: restrict

The restrict C keyword is a type qualifier. An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association requires that all accesses to that object use, directly or indirectly, the value of that particular pointer. The use of the restrict qualifier is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning.

The restrict qualifier can only be used for object pointers.

For example:

```
int * restrict p; // OK
int * restrict q; // OK
int restrict i;  // error
```

With these declarations you tell the C compiler that if an object is accessed using one of p or q, and that object is modified anywhere in the program, then it is never accessed through the other.

C/C++ keyword: return

Syntax

```
return;
return( value );
```

The return statement causes execution to jump from the current function to whatever function called the current function. An optional `value` can be returned. A function may have more than one return statement.

C/C++ keyword: short

The short keyword is a data type modifier that is used to declare short integer variables. See the C/C++ data types.

C/C++ keyword: signed

The signed keyword is a data type modifier that is usually used to declare signed char variables. See the C/C++ data types.

C/C++ keyword: sizeof

The sizeof operator is a compile-time operator that returns the size of the argument passed to it. The size is a multiple of the size of a `char`, which on many personal computers is 1 byte (or 8 bits). The number of bits in a `char` is stored in the `CHAR_BIT` constant defined in the `<limits.h>` header file.

For example, the following code uses sizeof to display the sizes of a number of variables:

```
struct EmployeeRecord
{
  int ID;
  int age;
  double salary;
  EmployeeRecord* boss;
};

...

printf("sizeof(int): %d\n", sizeof(int));
printf("sizeof(float): %d\n", sizeof(float));
printf("sizeof(double): %d\n", sizeof(double));
printf("sizeof(char): %d\n", sizeof(char));
printf("sizeof(EmployeeRecord): %d\n", sizeof(EmployeeRecord));
```

```
int i;
float f;
double d;
char c;
EmployeeRecord er;

printf("sizeof(i): %d\n", sizeof(i));
printf("sizeof(f): %d\n", sizeof(f));
printf("sizeof(d): %d\n", sizeof(d));
printf("sizeof(c): %d\n", sizeof(c));
printf("sizeof(er): %d\n", sizeof(er));
```

On some machines, the above code displays this output:

```
sizeof(int): 4
sizeof(float): 4
sizeof(double): 8
sizeof(char): 1
sizeof(EmployeeRecord): 20
sizeof(i): 4
sizeof(f): 4
sizeof(d): 8
sizeof(c): 1
sizeof(er): 20
```

Note that sizeof can either take a variable type (such as **int**) or a variable name (such as **i** in the example above).

It is also important to note that the sizes of various types of variables can change depending on what system you're on. Check out a description of the C data types for more information.

The parentheses around the argument are not required if you are using sizeof with a variable type (e.g. sizeof(int)).

C/C++ keyword: static

The static data type modifier is used to create permanent storage for variables. Static variables keep their value between function calls.

C/C++ keyword: static_cast

Syntax

```
TYPE static_cast<TYPE> (object);
```

The static_cast keyword can be used for any normal conversion between types. This includes any casts between numeric types, casts of pointers and references up the hierarchy, conversions with unary constructor, conversions with conversion operator. For conversions between numeric types no run-time checks are performed if data fits the new type. Conversion with unary constructor would be performed even if it is declared as explicit.

It can also cast pointers or references down and across the hierarchy as long as such conversion is avaliable and unambiguous. No run-time checks are performed.

C/C++ keyword: struct

Syntax

```
struct struct-name : inheritance-list
{
  public-members-list;
protected:
  protected-members-list;
private:
  private-members-list;
} object-list;
```

Structs are like `classes`, except that by default members of a struct are public rather than private. In C, structs can only contain data and are not permitted to have inheritance lists.

```
struct struct-name
{
  members-list;
} object-list;
```

The object list is optional - structs may be defined without actually instantiating any new objects.

For example, the following code creates a new data type called `Date` (which contains three integers) and also creates an instance of `Date` called `today`:

```
struct Date
{
  int day;
  int month;
  int year;
} today;

int main()
{
  today.day = 4;
  today.month = 7;
  today.year = 1776;
}
```

C/C++ keyword: switch

Syntax

```
switch( expression )
{
  case A:
    statement list;
    break;
  case B:
    statement list;
    break;
  ...
  case N:
    statement list;
    break;
  default:
    statement list;
    break;
}
```

The switch statement allows you to test an expression for many values, and is commonly used as a replacement for multiple if()...else if()...else if()... statements. break statements are required between each case statement, otherwise execution will "fall-through" to the next case statement. The default case is optional. If provided, it will match any case not explicitly covered by the preceding cases in the switch statement. For example:

```
char keystroke = getch();
switch( keystroke )
{
  case 'a':
  case 'b':
  case 'c':
  case 'd':
    KeyABCDPressed();
    break;
  case 'e':
    KeyEPressed();
    break;
  default:
    UnknownKeyPressed();
    break;
}
```

C/C++ keyword: template

Syntax

```
template <class data-type> return-type name( parameter-list )
{
  statement-list;
}
```

Templates are used to create generic functions and can operate on data without knowing the nature of that data. They accomplish this by using a placeholder data-type for which many other data types can be substituted.

For example, the following code uses a template to define a generic swap function that can swap two variables of any type:

```
template<class X> void genericSwap( X &a, X &b )
{
  X tmp;

  tmp = a;
  a = b;
  b = tmp;
}
int main(void)
{
  ...
  int num1 = 5;
  int num2 = 21;
  printf("Before, num1 is %d and num2 is %d\n", num1, num2);
  genericSwap( num1, num2 );
  printf("After, num1 is %d and num2 is %d\n", num1, num2);
  char c1 = 'a';
  char c2 = 'z';
  printf("Before, c1 is %c and c2 is %c\n", c1, c2);
  genericSwap( c1, c2 );
  printf("After, c1 is %c and c2 is %c\n", c1, c2);
  ...
  return( 0 );
}
```

C/C++ keyword: this

The this keyword is a pointer to the current object. All member functions of a class have a this pointer.

C/C++ keyword: throw

Syntax

```
try
{
  statement list;
}
catch( typeA arg )
{
  statement list;
}
catch( typeB arg )
{
  statement list;
}
...
catch( typeN arg )
{
  statement list;
}
```

The throw statement is part of the C++ mechanism for exception handling. This statement, together with the try and catch statements, gives programmers an elegant mechanism for error recovery.

You will generally use a try block to execute potentially error-prone code. Somewhere in this code, a throw statement can be executed, which will cause execution to jump out of the try block and into one of the catch blocks.

A

```
catch (...)
{
}
```

will catch any throw without considering what kind of object was thrown and without giving access to the thrown object.

Writing

```
throw
```

within a catch block will re throw what ever was caught.

Example

```
try
{
  printf("Before throwing exception\n");
  throw 42;
  printf("Shouldn't ever see this!\n");
}
catch( int error )
{
  printf("Error: caught exception %d\n", error);
}
```

C/C++ keyword: true

The Boolean value of "true".

C/C++ keyword: try

The try statement attempts to execute exception-generating code. See the throw statement for more details.

C/C++ keyword: typedef

Syntax

```
typedef existing-type new-type;
```

The typedef keyword allows you to create a new alias for an existing data type.

This is often useful if you find yourself using a unwieldy data type -- you can use typedef to create a shorter, easier-to-use name for that data type. For example:

```
typedef unsigned int* pui_t;
  // data1 and data2 have the same type
piu_t data1;
unsigned int* data2;
```

C/C++ keyword: typeid

Syntax

```
typeid( object );
```

The typeid operator returns a reference to a type_info object that describes *object*.

C/C++ keyword: typename

The typename keyword can be used to describe an undefined type or in place of the class keyword in a template declaration.

C/C++ keyword: union

Syntax

C++:

```
union union-name
{
  public-members-list;
private:
  private-members-list;
  members-list;
} object-list;
```

C:

```
union union-name
{
  members-list;
} object-list;
```

A union is like a class, except that all members of a union share the same memory location and are by default public rather than private. For example:

```
union Data
{
  int i;
  char c;
};
```

C/C++ keyword: unsigned

The unsigned keyword is a data type modifier that is usually used to declare unsigned int variables. See the C/C++ data types.

C/C++ keyword: using

The using keyword is used to import a namespace (or parts of a namespace) into the current scope.

Example

For example, the following code imports the entire `std` namespace into the current scope so that items within that namespace can be used without a preceeding "std::".

```
using namespace std;
```

Alternatively, the next code snippet just imports a single element of the `std` namespace into the current namespace:

```
using std::cout;
```

C/C++ keyword: virtual

Syntax

```
virtual return-type name( parameter-list );
virtual return-type name( parameter-list ) = 0;
```

The virtual keyword can be used to create virtual functions, which can be overridden by derived classes.

- A virtual function indicates that a function can be overridden in a subclass, and that the overridden function will actually be used.
- When a base object pointer points to a derived object that contains a virtual function, the decision about which version of that function to call is based on the type of object pointed to by the pointer, and this process happens at run-time.
- A base object can point to different derived objects and have different versions of the virtual function run.

If the function is specified as a pure virtual function (denoted by the = 0), it must be overridden by a derived class.

For example, the following code snippet shows how a child class can override a virtual method of its parent, and how a non-virtual method in the parent cannot be overridden:

```
class Base
{
public:
  void nonVirtualFunc()
  {
    printf("Base: non-virtual function\n");
  }
  virtual void virtualFunc()
  {
    printf("Base: virtual function\n");
  }
};
```

```
class Child : public Base
{
public:
  void nonVirtualFunc()
  {
    printf("Child: non-virtual function\n");
  }
  void virtualFunc()
  {
    printf("Child: virtual function\n");
  }
};

int main()
{
  Base* basePointer = new Child();
  basePointer->nonVirtualFunc();
  basePointer->virtualFunc();
  return 0;
}
```

When run, the above code displays:

```
Base: non-virtual function
Child: virtual function
```

C/C++ keyword: void

The void keyword is used to denote functions that return no value, or generic variables which can point to any type of data. Void can also be used to declare an empty parameter list. Also see the C/C++ data types.

C/C++ keyword: volatile

The volatile keyword is an implementation-dependent type qualifier, used when declaring variables, which prevents the compiler from optimizing those variables. Volatile should be used with variables whose value can change in unexpected ways (i.e. through an interrupt), which could conflict with optimizations that the compiler might perform.

C/C++ keyword: wchar_t

The keyword wchar_t is used to declare wide character variables. Also see the C/C++ data types.

C/C++ keyword: while

Syntax

```
while( condition )
{
  statement-list;
}
```

The while keyword is used as a looping construct that will evaluate the *statement-list* as long as *condition* is true. Note that if the *condition* starts off as false, the *statement-list* will never be executed. (You can use a do loop to guarantee that the statement-list will be executed at least once.) For example:

```
bool done = false;
while( !done )
{
  ProcessData();
  if( StopLooping() )
  {
    done = true;
  }
}
```

Processor Specific Keywords

Below is a list of processor specific C keywords. They do not belong to the standard C language. The implementation of a processor specific keyword may differ per processor. In this section they are explained separately per processor, though some keywords are the same for all processors.

	ARM	Micro Blaze	Nios II	Power PC	TSK51x/ TSK52x	TSK80x	TSK3000	
__asm()	x	x	x	x	x	x	x	use assembly instructions in C source
__at()	x	x	x	x	x	x	x	place data object at an absolute address
__frame()	x	x			x	x		safe registers for an interrupt function
__interrupt()	x	x	x	x	x	x	x	qualify function as interrupt service routine
__nesting_enabled	x							force save of LR and enable interrupts
__noinline	x	x	x	x	x	x	x	prevent compiler from inlining function
__noregaddr					x			register bank independent code generation
__novector	x							prevent compiler from generating vector symbol
__packed__	x	x	x	x			x	prevent alignment gaps in structures
__reentrant					x			qualify function as reentrant
__registerbank()					x			assign new register bank to interrupt function
__reset						x		jump to function at system reset
__static					x			qualify function as static
__system		x						qualify function as non-interruptable
__unaligned	x	x	x	x			x	suppress the alignment of objects or structure members

Processor specific keyword: __asm()

With the `__asm()` keyword you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

Processor specific keyword: __asm() (ARM)

Syntax

```
__asm( "instruction_template"
      [ : output_param_list
      [ : input_param_list
      [ : register_save_list]]] );
```

With the `__asm()` keyword you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*
%*parm_nr*[**.***regnum*]	Parameter number in the range 0 .. 9. With the optional **.***regnum* you can access an individual register from a register pair. For example, with the word register `R12`, `.0` selects register `R1`.
output_param_list	[[**"=**[**&**]*constraint_char***"(**C_expression**)**],...]
input_param_list	[[**"***constraint_char***"(**C_expression**)**],...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*.
C_expression	Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment.
register_save_list	[[**"***register_name***"**],...]
register_name:q	Name of the register you want to reserve.

Constraint	Type	Operand	Remark
R	general purpose register (64 bits)	r0 .. r11	Thumb mode r0 .. r7 Based on the specified register. A register pair is formed (64-bit). For example r0r1.
r	general purpose register	r0 .. r11, lr	Thumb mode r0 .. r7
i	immediate value	#value	
l	label	*label*	
m	memory label	*variable*	stack or memory operand, a fixed address
number	other operand	same as %*number*	Input constraint only. The *number* must refer to an output parameter. Indicates that %*number* and *number* are the same register. Use %*number*.0 and %*number*.1 to indicate the first and second half of a register pair when used in combination with R.

Processor specific keyword: __asm() (MicroBlaze)

Syntax

```
__asm( "instruction_template"
      [ : output_param_list
      [ : input_param_list
      [ : register_save_list]]] );
```

With the `__asm()` keyword you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*
%*parm_nr*[**.***regnum*]	Parameter number in the range 0 .. 9. With the optional **.***regnum* you can access an individual register from a register pair. For example, with the word register `R12`, `.0` selects register `R1`.
output_param_list	[[**"=**[**&**]*constraint_char***"(**C_expression**)**],...]
input_param_list	[[**"***constraint_char***"(**C_expression**)**],...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*.
C_expression	Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment.
register_save_list	[[**"***register_name***"**],...]
register_name:q	Name of the register you want to reserve.

Constraint	Type	Operand	Remark
r	general purpose register	r0 .. r31	
i	immediate value	#immval	
number	other operand	same as %*number*	Input constraint only. The *number* must refer to an output parameter. Indicates that %*number* and *number* are the same register. Use %*number*.0 and %*number*.1 to indicate the first and second half of a register pair when used in combination with R.

Processor specific keyword: __asm() (Nios II)

Syntax

```
__asm( "instruction_template"
      [ : output_param_list
      [ : input_param_list
      [ : register_save_list]]] );
```

With the `__asm()` keyword you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*
%*parm_nr*[**.***regnum*]	Parameter number in the range 0 .. 9. With the optional **.***regnum* you can access an individual register from a register pair. For example, with the word register `R12`, `.0` selects register `R1`.
output_param_list	[[**"=**[**&**]*constraint_char***"(**C_expression**)**],...]
input_param_list	[[**"***constraint_char***"(**C_expression**)**],...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*.
C_expression	Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment.
register_save_list	[[**"***register_name***"**],...]
register_name:q	Name of the register you want to reserve.

Constraint	Type	Operand	Remark
R	general purpose register (64 bits)	r0 .. r31	Based on the specified register, a register pair is formed (64-bit). For example r0:r1.
r	general purpose register (32 bits)	r0 .. r31	
i	immediate value	#value	
l	label	*label*	
m	memory label	*variable*	stack or memory operand, a fixed address
number	other operand	same as %*number*	Input constraint only. The *number* must refer to an output parameter. Indicates that %*number* and *number* are the same register. Use %*number*.0 and %*number*.1 to indicate the first and second half of a register pair when used in combination with R.

Processor specific keyword: __asm() (PowerPC)

Syntax

```
__asm( "instruction_template"
        [ : output_param_list
        [ : input_param_list
        [ : register_save_list]]] );
```

With the `__asm()` keyword you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*
%*parm_nr*[**.***regnum*]	Parameter number in the range 0 .. 9. With the optional **.***regnum* you can access an individual register from a register pair. For example, with the word register `R12`, `.0` selects register `R1`.
output_param_list	[[**"=**[**&**]*constraint_char***"(**C_expression**)**],...]
input_param_list	[[**"***constraint_char***"(**C_expression**)**],...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*.
C_expression	Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment.
register_save_list	[[**"***register_name***"**],...]
register_name:q	Name of the register you want to reserve.

Constraint	Type	Operand	Remark
R	general purpose register (64 bits)	%r0..%r31	Based on the specified register, a register pair is formed (64-bit). For example %r0:%r1.
r	general purpose register (32 bits)	%r0..%r31	
i	immediate value	*#value*	
l	label	*label*	
m	memory label	*variable*	stack or memory operand, a fixed address
number	other operand	same as *number*	Input constraint only. The *number* must refer to an output parameter. Indicates that %*number* and *number* are the same register. Use %*number*.0 and %*number*.1 to indicate the first and second half of a register pair when used in combination with R.

Processor specific keyword: __asm() (TSK3000)

Syntax

```
__asm( "instruction_template"
       [ : output_param_list
       [ : input_param_list
       [ : register_save_list]]] );
```

With the `__asm()` keyword you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*
%*parm_nr*[**.***regnum*]	Parameter number in the range 0 .. 9. With the optional **.***regnum* you can access an individual register from a register pair. For example, with the word register `R12`, `.0` selects register `R1`.
output_param_list	[[**"=**[**&**]*constraint_char***"(**C_expression**)**],...]
input_param_list	[[**"***constraint_char***"(**C_expression**)**],...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*.
C_expression	Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment.
register_save_list	[[**"***register_name***"**],...]
register_name:q	Name of the register you want to reserve.

Constraint	Type	Operand	Remark
R	general purpose register (64 bits)	$v0,$v1, $a0 .. $a3, $kt0, $kt1, $t0..$t9, $s0 .. $s8	Based on the specified register, a register pair is formed (64-bit). For example $v0:$v1.
r	general purpose register (32 bits)	$v0,$v1, $a0 .. $a3, $kt0, $kt1, $t0..$t9, $s0 .. $s8	
i	immediate value	*#value*	
l	label	*label*	
m	memory label	*variable*	stack or memory operand, a fixed address
H	multiply and devide register higher result	$hi	
L	multiply and devide register lower result	$lo	
number	other operand	same as *%number*	Input constraint only. The *number* must refer to an output parameter. Indicates that *%number* and *number* are the same register.

Processor specific keyword: __asm() (TSK51x/TSK52x)

Syntax

```
__asm( "instruction_template"
       [ : output_param_list
       [ : input_param_list
       [ : register_save_list]]] );
```

With the `__asm()` keyword you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

instruction_template	Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr*
%*parm_nr*[**.***regnum*]	Parameter number in the range 0 .. 9. With the optional **.***regnum* you can access an individual register from a register pair. For example, with the word register `R12`, `.0` selects register `R1`.
output_param_list	[[**"=**[**&**]*constraint_char***"(**C_expression**)**],...]
input_param_list	[[**"***constraint_char***"(**C_expression**)**],...]
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*.
C_expression	Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment.
register_save_list	[[**"***register_name***"**],...]
register_name:q	Name of the register you want to reserve.

Constraint	Type	Operand	Remark
a	accumulator	A	
b	bit	ACC.[0..7], B.[0..7], C, AC, F0, RS1, RS0, OV, F1, P, _bitvar	bit registers/variables
d	direct register	PSW, SP, B, ACC, DPH, DPL, AR[0..7]	direct address of registers
i	immediate value	*#data*, *#data16*	
m	memory	*direct*, *label*, *addr11*, *addr16*, *rel*	memory variable or function address
p	data page pointer	DPTR	
r	register	R[0..7]	
R	registers	R01, R12, R23, R34, R45, R56, R67	word registers
s	register indirect	@R0, @R1	register indirect addressing
number	other operand	same as **%***number*	Input constraint only. The *number* must refer to an output parameter. Indicates that %*number* and *number* are the same register.

Processor specific keyword: __asm() (TSK80x)

Syntax

```
__asm( "instruction_template" );
```

With the `__asm()` keyword you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

instruction_template One or more TSK80x assembly instructions

Constraint	Type	Operand	Remark
i	immediate value	*#value*	
m	memory	*address*, *label*	stack or memory operand, a fixed address or indexed addressing
r	register	A, B, C, D, E, H, L, I, R IX, IY, SP, AF, BC, DE, HL	8-bit register 16-bit register
number	other operand	same as **%***number*	Input constraint only. The *number* must refer to an output parameter. Indicates that %*number* and *number* are the same register.

Processor specific keyword: __at() (all processors)

Syntax

```
int myvar __at(address);
```

With the attribute `__at()` you can place an object at an absolute address.

Example

```
unsigned char Display[80*24] __at(0x2000 );
```

The array `Display` is placed at address 0x2000. In the generated assembly, an absolute section is created. On this position space is reserved for the variable `Display`.

```
int i __at(0x1000) = 1;
```

The variable `i` is placed at address 0x1000 and is initialized at 1.

```
void f(void) __at( 0xf0ff + 1 ) { }
```

The function `f` is placed at address 0xf100.

Take note of the following restrictions if you place a variable at an absolute address:

- The argument of the `__at()` attribute must be a constant address expression.
- You can place only global variables at absolute addresses. Parameters of functions, or automatic variables within functions cannot be placed at absolute addresses.
- When declared `extern`, the variable is not allocated by the compiler. When the same variable is allocated within another module but on a different address, the compiler, assembler or linker will not notice, because an assembler external object cannot specify an absolute address.
- When the variable is declared `static`, no public symbol will be generated (normal C behavior).
- You cannot place structure members at an absolute address.
- Absolute variables cannot overlap each other. If you declare two absolute variables at the same address, the assembler and / or linker issues an error. The compiler does not check this.
- When you declare the same absolute variable within two modules, this produces conflicts during link time (except when one of the modules declares the variable 'extern').
- If you use 0 as an address, this value is ignored. A zero value indicates a relocatable section.

Processor specific keyword: __frame()

With the function type qualifier `__frame()` you can specify which registers and SFRs must be saved for a particular interrupt function. Only the specified registers will be pushed and popped from the stack.
If you do not specify the function qualifier `__frame()`, the C compiler determines which registers must be pushed and popped.

Processor specific keyword: __frame() (ARM)

Syntax

```
void __interrupt_xxx __frame(reg[, reg]...) isr( void )
{
  ...
}
```

With the function type qualifier `__frame()` you can specify which registers and SFRs must be saved for a particular interrupt function. *reg* can be any register defined as an SFR. Only the specified registers will be pushed and popped from the stack.
If you do not specify the function qualifier `__frame()`, the C compiler determines which registers must be pushed and popped.

Example

```
__interrupt_irq __frame(R4,R5,R6) void alarm( void )
{
  ...
}
```

Processor specific keyword: __frame() (MicroBlaze)

Syntax

```
void __interrupt(vector_address) __frame(reg[, reg]...) isr( void )
{
  ...
}
```

With the function type qualifier `__frame()` you can specify which registers and SFRs must be saved for a particular interrupt function. *`reg`* can be any register defined as an SFR. Only the specified registers will be pushed and popped from the stack.
If you do not specify the function qualifier `__frame()`, the C compiler determines which registers must be pushed and popped.

Example

```
__interrupt_irq __frame(R4,R5,R6) void alarm( void )
{
  ...
}
```

Processor specific keyword: __frame() (TSK51x/TSK52x)

Syntax

```
void __interrupt(vector_address[, vector_address]...)
    __frame(reg[, reg]...) isr( void )
    {
  ...
}
```

With the function type qualifier `__frame()` you can specify which registers and SFRs must be saved for a particular interrupt function. *`reg`* can be any register defined as an SFR. Only the specified registers will be pushed and popped from the stack. If you do not specify the function qualifier `__frame()`, the C compiler determines which registers must be pushed and popped.

Example

```
__interrupt( 0x10 ) __frame(A,R0,R1) void alarm( void )
{
  ...
}
```

Processor specific keyword: __frame() (TSK80x)

Syntax

```
void __interrupt(vector_address[, vector_address]...)
    __frame(reg[, reg]...) isr( void )
    {
  ...
}
```

With the function type qualifier `__frame()` you can specify which registers and SFRs must be saved for a particular interrupt function. *reg* can be any register defined as an SFR. Only the specified registers will be pushed and popped from the stack. If you do not specify the function qualifier `__frame()`, the C compiler determines which registers must be pushed and popped.

Example

```
__interrupt( 0x10 ) __frame(A,R0,R1) void alarm( void )
{
  ...
}
```

Processor specific keyword: __interrupt

The TASKING C compiler supports a number of function qualifiers and keywords to program interrupt service routines (ISR). An *interrupt service routine* (or: *interrupt function*, *interrupt handler*, *exception handler*) is called when an interrupt event (or: *service request*) occurs.

Processor specific keyword: __interrupt() (MicroBlaze)

Syntax

```
void __interrupt(vector_address) isr( void )
{
  ...
}
```

With the function type qualifier `__interrupt()` you can declare a function as an interrupt service routine. The function type qualifier `__interrupt()` takes one vector address as argument. With the `__interrupt()` keyword, a jump to the actual interrupt handler is caused.

Interrupt functions cannot return anything and must have a void argument type list.

The MicroBlaze supports five types of exceptions. The next table lists the types of exceptions and the processor mode that is used to process that exception. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the exception vectors.

Exception type	Vector address	Return register	Return instruction	Function type qualifier
Reset	0x00000000	-	-	__interrupt(0x00000000)
User vector (exception)	0x00000008	r15	rtsd	__interrupt(0x00000008)
Interrupt	0x00000010	r14	rtid	__interrupt(0x00000010)
Break	0x00000018	r16	rtbd	__interrupt(0x00000018)
Hardware exception	0x00000020	r17	rted	__interrupt(0x00000020)

Example

```
void __interrupt( 0x00000008 ) my_handler( void )
{
  ...
}
```

Processor specific keyword: __interrupt (Nios II)

Syntax

```
void __interrupt isr( void )  __at(exception_address)
{
  ...
}
```

With the function type qualifier `__interrupt` you can declare a function as an interrupt service routine. You can specify the interrupt jump location (exception address) with the attribute `__at()`. Note that this address is determined by your hardware.

Interrupt functions cannot return anything and must have a void argument type list.

Example

```
void __interrupt my_handler( void ) __at(0x20)
{
  ...
}
```

Processor specific keyword: __interrupt() (TSK3000)

Syntax

```
void __interrupt(vector_number[, vector_number]...) isr( void )
{
  ...
}
```

With the function type qualifier `__interrupt()` you can declare a function as an interrupt service routine. The function type `qualifier __interrupt()` takes one or more vector numbers (0..31) as argument(s). All supplied vector numbers will be initialized to point to the interrupt function.

Interrupt functions cannot return anything and must have a void argument type list.

Example

```
void __interrupt( 7 ) serial_receive( void )
{
  ...
}
```

Processor specific keyword: __interrupt() (TSK51x/TSK52x)

Syntax

```
void __interrupt(vector_address[, vector_address]...) isr( void )
{
  ...
}
```

With the function type qualifier `__interrupt()` you can declare a function as an interrupt service routine. The function type qualifier `__interrupt()` takes one or more vector addresses as argument(s). All supplied vector addresses will be initialized to point to the interrupt function.

Interrupt functions cannot return anything and must have a void argument type list.

Example

```
void __interrupt(vector_address[, vector_address]...) isr( void )
{
  ...
}
```

Note: If you want to use interrupt numbers instead of vector addresses for the TSK51A core, you can use the `__INTNO` macro which is defined in the delivered special function register file (regtsk51a.sfr) as:

Processor specific keyword: __interrupt(), __interrupt_indirect() (PowerPC)

Syntax

```
void __interrupt(vector_number) isr( void )
{
  ...
}

void __interrupt_indirect(vector_number[, vector_number]...) isr( void )
{
  ...
}
```

You can define two types of interrupt service routines

`__interupt()`	Fastest interrupt service routine. The interrupt function will be placed directly at the interrupt vector, saving a jump instruction.
`__interrupt_indirect()`	More flexible interrupt service routine. Takes one or more vector numbers as arguments. The vector table contains a jump to the actual interrupt handler. Useful when you want to use the same interrupt function for different interrupts.

The interrupt number you specify for `__interrupt()` or `__interrupt_indirect()`must be in the range 0 to 15 (inclusive).

Interrupt functions cannot return anything and must have a void argument type list.

Example

```
void __interrupt(5) external_interrupt( void )
{
  /* function code fits at interrupt vector */
}

void __interrupt_indirect( 7 ) serial_receive( void )
{
  ...
}
```

Note: Interrupts of the PowerPC are divided into two classes: *critical* and *non-critical* interrupts. You cannot simultaneously specify the vector numbers of both critical and non-critical functions in the argument list of the `__interrupt_indirect()` keyword.

Processor specific keyword: __interrupt_nmi, __interrupt_mode1, __interrupt_mode2 (TSK80x)

Syntax

```
void __interrupt(vector_address[, vector_address]...) isr( void )
{
  ...
}

void __interrupt_mode2(vector_address[, vector_address]...) isr( void )
{
  ...
}
```

You can define three types of interrupt service routines with the following function type qualifiers:

`__interupt_nmi`	Non-maskable interrupt: cannot be disabled by program control Jumps to the first instruction at 0x66
`__interupt_mode1`	Similar to a non-maskable interrupt: a normal interrupt acknowledge cycle is made, but data put onto the data bus is ignored. Jumps to the first instruction at 0x38
`__interupt()` `__interupt_mode2()`	Most flexible interrupt. Takes one or more vector addresses as argument. Jumps to the first instruction at the specified vector address.

The function qualifiers `__interrupt()` and `__interrupt_mode2()` take one or more vector addresses as argument(s). All supplied vector addresses will be initialized to point to the interrupt function.
Interrupt functions cannot return anything and must have a void argument type list.

Example

TSK80x non-maskable interrupt:

```
void __interrupt_nmi IntNmHandler( void )
{
  InterruptNm();
}
```

TSK80x mode 1 interrupt:

```
void __interrupt_mode1 IntMode1Handler( void )
{
  InterruptM1();
}
```

TSK80x mode 2 interrupt (variant 1):

```
void __interrupt( 0x10 ) IntMode2Handler( void )
{
  InterruptM2();
}
```

TSK80x mode 2 interrupt (variant 2):

```
void __interrupt_mode2( 0x10 ) IntHandler( void )
{
  InterruptM2();
}
```

This will reserve a word (`.dw` directive) on address 0x10, where the address of the interrupt function is placed.

Processor specific keyword: __interrupt_und, __interrupt_svc, __interrupt_iabt, __interrupt_dabt, __interrupt_irq, __interrupt_fiq, __interrupt() (ARM)

Syntax

```
void __interrupt_xxx  isr( void )
{
  ...
}
void __interrupt(n) isr( void )
{
  ...
}
```

You can define six types of exception handlers with the function type qualifiers:

- `__interrupt_und`
- `__interrupt_svc`
- `__interrupt_iabt`
- `__interrupt_dabt`
- `__interrupt_irq`
- `__interrupt_fiq`

You can also use the general `__interrupt()` function qualifier.

Interrupt functions and other exception handlers cannot return anything and must have a void argument type list.

The ARM supports seven types of exceptions. The next table lists the types of exceptions and the processor mode that is used to process that exception. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the exception vectors.

Exception type	Mode	Normal address	High vector address	Function type qualifier
Reset	Supervisor	0x00000000	0xFFFF0000	
Undefined instructions	Undefined	0x00000004	0xFFFF0004	`__interrupt_und`
Supervisor call (software interrupt)	Supervisor	0x00000008	0xFFFF0008	`__interrupt_svc`
Prefetch abort	Abort	0x0000000C	0xFFFF000C	`__interrupt_iabt`
Data abort	Abort	0x00000010	0xFFFF0010	`__interrupt_dabt`
IRQ (interrupt)	IRQ	0x00000018	0xFFFF0018	`__interrupt_irq`
FIQ (fast interrupt)	FIQ	0x0000001C	0xFFFF001C	`__interrupt_fiq`

Example

```
void __interrupt_irq serial_receive( void )
{
  ...
}
```

Processor specific keyword: __nesting_enabled (ARM)

Syntax

```
__interrupt_xxx __nesting_enabled isr( void )
{
  ...
}
```

Normally interrupts are disabled when an exception handler is entered. With the function qualifier `__nesting_enabled` you can force that the link register (LR) is saved and that interrupts are enabled.

Example

```
void __interrupt_svc __nesting_enabled svc(int n )
{
  if ( n == 2 )
  {
    __svc(3);
  }
}
```

Note: The function qualifier `__nesting_enabled` is not available for M-profile architectures.

Processor specific keyword: __noinline (all processors)

Syntax

```
__noinline int functionA( int i )
{
    // do not inline this function
}
```

With the `__noinline` keyword, you prevent a function from being inlined, regardless of the optimization settings.

Example

```
__noinline unsigned int abs(int val)
{
  unsigned int abs_val = val;
  if (val < 0) abs_val = -val;
  return abs_val;
}
```

Processor specific keyword: __novector (ARM)

Syntax

```
__interrupt_xxx __novector isr( void )
{
  ...
}
```

You can prevent the compiler from generating the `__vector_`*n* you can symbol by specifying the function qualifier `__novector`. This can be necessary if you have more than one interrupt handler for the same exception, for example for different IRQ's or for different run-time phases of your application. Without the `__novector` function qualifier the compiler generates the `__vector_`*n* symbol multiple times, which results in a link error.

Example

```
void __interrupt_irq __novector another_handler(void)
{
  /* used __novector to prevent multiple _vector_6 symbols */
}
```

Processor specific keyword: __noregaddr (TSK51x/TSK52x)

You can use the keyword `__noregaddr` to switch to register bank independent code generation. In order to generate very efficient code the compiler uses absolute register addresses in its code generation. For example a register to register 'move'. Since there is no 'MOV register, register' instruction, the compiler will generate a 'MOV register, direct' with the absolute address of the source register as the second operand.

The absolute address of a register depends on the register bank, but sometimes this dependency is undesired. For example when a function is called from both the main thread and an interrupt thread. If both threads use different register banks, they cannot call a function that uses absolute register addresses. To overcome this, you can instruct the compiler to generate a register bank independent function that can be called from both threads.

Example

```
__noregaddr int func( int x )
{
  /* this function can be called from any function
    independent of its register bank */
  return x+1;
}

__registerbank(1) void f1( void )
{
  func( 1 );
}

__registerbank(0) void main( void )
{
  func( 0 );
}
```

Processor specific keyword: __packed__ (all 32-bit processors)

To prevent alignment gaps in structures, you can use the attribute `__packed__`. When you use the attribute `__packed__` directly after the keyword `struct`, all structure members are marked `__unaligned`.

Example

The following two declarations are the same:

```
struct __packed__
{
  char c;
  int i;
} s1;

struct
{
  __unaligned char c;
  __unaligned int i;
} s2;
```

The attribute `__packed__` has the same effect as adding the type qualifier `__unaligned` to the declaration to suppress the standard alignment.
You can also use `__packed__` in a pointer declaration. In that case it affects the alignment of the pointer itself, not the value of the pointer. The following two declarations are the same:

```
int * __unaligned p;
int * p __packed__;
```

Processor specific keyword: __registerbank() (TSK51x/TSK52x)

Syntax

```
void __interrupt(vector_address[, vector_address]...)
    __registerbank(bank) isr( void )
    {
  ...
}
```

For the TSK51x/TSK52x it is possible to assign a new register bank to an interrupt function, which can be used on the processor to minimize the interrupt latency because registers do not need to be pushed on stack. You can switch register banks with the `__registerbank()` function qualifier.

When you specify the __registerbank() qualifier the registers R0-R7 are implicitly saved when the register bank is being switched (by using the predefined symbolic register addresses AR0-AR7).

The default register bank used is bank 0.

Example

Suppose timer(), from the previous example, is calling get_number(). The function prototype (and definition) of get_number() should contain the correct __registerbank().

```
#define __INTNO(nr) ((8*nr)+3)

__interrupt(__INTNO(1)) __registerbank(2) void timer(void);
```

The TSK51x/TSK52x compiler places a long-jump instruction on the vector address 11 of interrupt number 1, to the `timer()` routine, which switches the register bank to bank 2 and saves some more registers. When `timer()` is completed, the extra registers are popped, the bank is switched back to the original value and a `RETI` instruction is executed.

You can call another C function from the interrupt C function. However, this function must be compiled with the same `__registerbank(bank-nr)` qualifier, because the compiler generates code which uses the addresses of the registers R0-R7. Therefore, the `__registerbank(bank-nr)` qualifier is also possible with normal C functions (and their prototype declarations).

Suppose `timer()`, from the previous example, is calling `get_number()`. The function prototype (and definition) of `get_number()` should contain the correct `__registerbank()`.

```
__registerbank( 2 ) int get_number( void );
```

The compiler checks if a function calls another function using another register bank, which is an error.

Processor specific keyword: __reset (TSK80x)

The function qualifier `__reset` generates a jump located at address 0x00 (system reset) to the location of the function (in the example `_start()`). This way it is possible to execute a piece of code at system reset.

Example

```
void __reset _start(void)
{
  __setsp((unsigned int)_lc_es-1);  /* initialize stack pointer */
  __init();                          /* initialize C variables  */
  exit( main(0) )                    /* argc is 0                */
}
```

Processor specific keyword: __static, __reentrant (TSK51x/TSK52x)

You can use the function qualifiers `__static` or `__reentrant` to specify a function as static or reentrant, respectively.

If you do not specify a function qualifier, the TSK51x/TSK52x compiler assumes that those functions are static. In static functions parameters and automatic variables are not allocated on a stack, but in a static area. Reentrant functions use a less efficient virtual dynamic stack which allows you to call functions recursively.

Example

```
void f_static( void )
{
  /* this function is by default __static */
}
__reentrant int f_reentrant ( void )
{
  int i;    /* variable i is placed on a virtual stack */
}
```

Processor specific keyword: __system (MicroBlaze)

You can use the function qualifier `__system` to specify a function that cannot be interrupted. A function defined with the `__system` qualifier is called with a `BRK` or `BRKI` instruction (instead of a branch instruction) and returns with a `RTBD` instruction (instead of the `RTS` or `RTSD` instruction).

You cannot use the function qualifier `__system` on interrupt functions. Functions defined with `__system` cannot be inlined.

Example

```
__system void non_interruptable( int a, int b )
{
  ...
}
```

Processor specific keyword: __unaligned (all 32-bit processors)

With the type qualifier `__unaligned` you can specify to suppress the alignment of objects or structure members. This can be useful to create compact data structures. In this case the alignment will be one bit for bit-fields or one byte for other objects or structure members.
At the left side of a pointer declaration you can use the type qualifier `__unaligned` to mark the pointer value as potentially unaligned. This can be useful to access externally defined data. However the compiler can generate less efficient instructions to dereference such a pointer, to avoid unaligned memory access.

Example

```
struct
 {
   char c;
   __unaligned int i;   /* aligned at offset 1 ! */
 } s;

 __unaligned int * up = & s.i;
```

Intrinsic functions

The following is a list of all intrinsic functions. Intrinsic function do not belong to the standard C language but the compiler may support intrinsics for a specific processor.

	ARM	Micro Blaze	Nios II	Power PC	TSK51x/ TSK52x	TSK80x	TSK3000	
__alloc	x	x	x	x	x	x	x	Allocate memory
__break							x	Insert break instruction
__dotdotdot__		x	x		x	x		Variable argument '...' operator
__free	x	x	x	x	x	x	x	Deallocate memory
__getbit					x			Get the value of a bit
__putbit					x			Set the value of a bit
__get_return_address	x	x	x	x	x		x	Function return address (when profiling)
__getapsr	x							Get APSR status register
__setapsr	x							Set APSR status register
__getcpsr	x							Get CPSR status register
__getipsr	x							Get IPSR status register
__setcpsr	x							Set CPSR status register
__getspsr	x							Get SPSR status register
__setspsr	x							Set SPSR status register
__cgetfsl		x						Read control words from fast simplex link
__cputfsl		x						Write control words to fast simplex link
__getfsl		x						Read data words from fast simplex link
__putfsl		x						Write data words to fast simplex link
__getfsr		x						Get FSR register
__putfsr		x						Set FSR register
__getmsr		x						Get MSR register
__putmsr		x						Set MSR register
__msrclr		x						Clear bits in MSR register
__msrset		x						Set bits in MSR register
__getpc		x						Get value of program counter PC
__mfspr				x				Get special function register
__mtspr				x				Set special function register
__mfctr				x				Get special function register CTR
__mtctr				x				Set special function register CTR
__mflr				x				Get special function register LR
__mtlr				x				Set special function register LR
__mfmsr				x				Get special function register MSR
__mtmsr				x				Set special function register MSR
__mfxer				x				Get special function register XER

__mtxer			x				Set special function register XER
__getsp					x		Get stack pointer (SP)
__setsp					x		Set stack pointer (SP)
__mfc0						x	Get value from SPR of coprocessor 0
__mtc0						x	Set value to SPR of coprocessor 0
__nop	x	x		x		x	Insert NOP instruction
__rol				x			Rotate left
__ror				x			Rotate right
__svc	x						Generate software interrupt.
__testclear				x			Read and clear semaphore
__vsp__				x			Virtual Stack Pointer in use

Intrinsic function: __alloc

Syntax

```
void * volatile __alloc( __size_t size );
```

Allocate memory. Same as library function malloc().

Returns: a pointer to space in external memory of size bytes length. NULL if there is not enough space left.

Intrinsic function: __break

Syntax

```
volatile int __break(int val);
```

Generates the assembly break instruction. *Val* is a 20-bit value which will be encoded in the code field of the break instruction..

Returns: nothing.

Intrinsic function: __cgetfsl (MicroBlaze)

Syntax

```
_Bool volatile __cgetfsl( unsigned char channel,
                          unsigned int * ctrl, _Bool wait );
```

Read control words from a specified fast simplex link (fsl) channel.

Returns: True if valid data was read from the specified channel, otherwise False.

Intrinsic function: __cputfsl (MicroBlaze)

Syntax

```
_Bool volatile __cputfsl( unsigned char channel,
                          unsigned int * ctrl, _Bool wait );
```

Write control words to a specified fast simplex link (fsl) channel.

Returns: True if valid data was read from the specified channel, otherwise False.

Intrinsic function: __dotdotdot__

Syntax

```
char * __dotdotdot__( void );
```

Variable argument '...' operator. Used in library function va_start().

Returns: the stack offset to the variable argument list.

Intrinsic function: __dotdotdot__ (Nios II)

Syntax

```
void * __dotdotdot__( void );
```

Variable argument '...' operator. Used in library function va_start().

Returns: the stack offset to the variable argument list.

Intrinsic function: __free

Syntax

```
void volatile __free( void *p );
```

Deallocates the memory pointed to by p. p must point to memory earlier allocated by a call to __alloc(). Same as library function free().

Returns: nothing.

Intrinsic function: __get_return_address

Syntax

```
__codeptr volatile __get_return_address( void );
```

Used by the compiler for profiling when you compile with the **-p (--profile)** option.

Returns: return address of a function.

Intrinsic function: __getapsr (ARM)

Syntax

```
unsigned int volatile __getapsr( void );
```

Note: This intrinsic is only available for ARMv6-M and ARMv7-M (M-profile architectures).

Get the value of the APSR status register.

Returns: the value of the status register APSR.

Intrinsic function: __getbit (TSK51x/TSK52x)

Syntax

```
__bit __getbit( bitaddressable, bitoffset );
```

Get the value of a bit. *`bitoffset`* must be an integral constant expression.

Returns: the bit at *`bitoffset`* (range 0-7 for a `char`, 0-15 for an int or 0-31 for a long) of the *`bitaddressable`* operand for use in bit expressions.

Example

```
__bdata unsigned char byte;
int i;

if ( __getbit( byte, 3 ) )
    i = 1;
```

Intrinsic function: __getcpsr (ARM)

Syntax

```
unsigned int volatile __getcpsr( void );
```

Get the value of the CPSR status register.

Returns: the value of the status register CPSR.

Intrinsic function: __getfsl (MicroBlaze)

Syntax

```
_Bool volatile __getfsl( unsigned char channel,
                         unsigned int * data, _Bool wait );
```

Read data words from a specified fast simplex link (fsl) channel. *`Channel`* must be a constant value in the range 0..7. The read data is stored in `*data`. With the boolean *`wait`* you can choose whether or not to wait for information: True: wait for information, False: do not wait for information (the channel may not provide data).

Returns: True if valid data was read from the specified channel, otherwise False.

Intrinsic function: __getfsr (MicroBlaze)

Syntax

```
unsigned int volatile __getfsr( void );
```

Get the value of the floating-point state register FSR.

Returns: the value of the floating-point state register FSR.

Intrinsic function: __getipsr (ARM)

Syntax

```
unsigned int volatile __getipsr( void );
```

Note: This intrinsic is only available for ARMv6-M and ARMv7-M (M-profile architectures).

Get the value of the IPSR status register.

Returns: the value of the status register IPSR.

Intrinsic function: __getmsr (MicroBlaze)

Syntax

```
unsigned int volatile __getmsr( void );
```

Get the value of the machine state register MSR.

Returns: the value of the machine state register MSR.

Intrinsic function: __getpc (MicroBlaze)

Syntax

```
unsigned int volatile __getpc( void );
```

Get the value of the program counter PC.

Returns: the value of the program counter.

Intrinsic function: __getsp (TSK80x)

Syntax

```
unsigned int volatile __getsp( void );
```

Get the value of the stack pointer SP.

Returns: the value of the stack pointer.

Intrinsic function: __getspsr (ARM)

Syntax

```
unsigned int volatile __getspsr( void );
```

Get the value of the SPSR status register.

Returns: the value of the status register SPSR.

Example

```
#define SR_F 0x00000040
#define SR_I 0x00000080

i = __setspsr (0, SR_F | SR_I);
if (i & (SR_F | SR_I))
{
  exit (6);    /* Interrupt flags not correct */
  }

if (__getspsr () & (SR_F | SR_I))
{
  exit (7);    /* Interrupt flags not correct */
  }
```

Intrinsic function: __mfc0 (TSK3000)

Syntax

```
volatile int __mfc0(int spr);
```

Get the value from special function register *spr* of coprocessor 0.

Returns: the value of the SPR register of coprocessor 0.

Intrinsic function: __mfctr (PowerPC)

Syntax

```
volatile int __mfctr(void);
```

Get the value from special function register CTR. (This equivalent to `__mfspr(0x009)`)

Returns: the value of the CTR register.

Intrinsic function: __mflr (PowerPC)

Syntax

```
volatile int __mflr(void);
```

Get the value from special function register LR. (This equivalent to `__mfspr(0x008)`)

Returns: the value of the LR register.

Intrinsic function: __mfmsr (PowerPC)

Syntax

```
volatile int __mfmsr(void);
```

Get the value from special function register MSR.

Returns: the value of the MSR register.

Intrinsic function: __mfspr (PowerPC)

Syntax

```
volatile int __mfspr(int spr);
```

Get the value from a special function register. `spr` is the number of the special purpose register and can be specified either as a decimal number or as a hexadecimal number.

Returns: the value of the specified special purpose register.

Intrinsic function: __mfxer (PowerPC)

Syntax

```
volatile int __mfxer(void);
```

Get the value from special function register XER. (This equivalent to `__mfspr(0x001)`)

Returns: the value of the XER register.

Intrinsic function: __msrclr (MicroBlaze)

Syntax

```
unsigned int __msrclr( unsigned int value );
```

Clear a number of bits in the machine state register MSR. `Value` should be a 14 bit mask. If you specify a value larger than 2^{14}, the instruction is ignored and the compiler will use the `getmsr` and `putmsr` instructions instead.

Returns: the value of the MSR register before bits were cleared.

Intrinsic function: __msrset (MicroBlaze)

Syntax

```
unsigned int __msrset( unsigned int value );
```

Set a number of bits in the machine state register MSR. `Value` should be a 14 bit mask. If you specify a value larger than 2^{14}, the instruction is ignored and the compiler will use the `getmsr` and `putmsr` instructions instead.

Returns: the value of the MSR register before bits were set.

Intrinsic function: __mtc0 (TSK3000)

Syntax

```
volatile void __mtc0(int val, int spr);
```

Put a value `val` into special purpose register `spr` of coprocessor 0.

Returns: nothing.

Intrinsic function: __mtctr (PowerPC)

Syntax

```
volatile void __mtctr(int val);
```

Put a value *val* into special function register CTR. (This equivalent to `__mtspr(0x009,val)`)

Returns: nothing.

Intrinsic function: __mtlr (PowerPC)

Syntax

```
volatile void __mtlr(int val);
```

Put a value *val* into special function register LR. (This equivalent to `__mtspr(0x008,val)`)

Returns: nothing.

Intrinsic function: __mtmsr (PowerPC)

Syntax

```
volatile void __mtmsr(int val);
```

Put a value *val* into special function register MSR.

Returns: nothing.

Intrinsic function: __mtspr (PowerPC)

Syntax

```
volatile void __mtspr(int spr, int val);
```

Put a value into a special function register. *spr* is the number of the special purpose register and can be specified either as a decimal number or as a hexadecimal number. *val* is the value to put into the specified register.

Returns: nothing.

Intrinsic function: __mtxer (PowerPC)

Syntax

```
volatile void __mtxer(int val);
```

Put a value *val* into special function register XER. (This equivalent to `__mtspr(0x001,val)`)

Returns: nothing.

Intrinsic function: __nop

Syntax

```
void __nop( void );
```

Generate NOP instructions.

Returns: nothing.

Example

```
__nop();          /* generate NOP instruction */
```

Intrinsic function: __putbit (TSK51x/TSK52x)

Syntax

```
void __putbit( __bit value, bitaddressable, bitoffset );
```

Assign a *value* to the bit at *bitoffset* (range 0-7 for a `char` , 0-15 for an int or 0-31 for a long) of the *bitaddressable* operand. *bitoffset* must be an integral constant expression.

Returns: nothing.

Example

```
__bdata unsigned int word;

__putbit( 1, word, 11 );
__putbit( 0, word, 10 );
```

Intrinsic function: __putfsl (MicroBlaze)

Syntax

```
_Bool volatile __putfsl( unsigned char channel,
                        unsigned int * data, _Bool wait );
```

Write data words to a specified fast simplex link (fsl) channel. *Channel* must be a constant value in the range 0..7. The data to write must be stored in `*data`. With the boolean *wait* you can choose whether or not to wait for information: True: wait for information, False: do not wait for information (the channel may not accept data).

Returns: True if valid data was written to the specified channel, otherwise False.

Intrinsic function: __putfsr (MicroBlaze)

Syntax

```
void volatile __putfsr( unsigned int value );
```

Set the *value* of the floating-point state register FSR to value.

Returns: nothing.

Intrinsic function: __putmsr (MicroBlaze)

Syntax

```
void volatile __putmsr( unsigned int value );
```

Set the value of the machine state register MSR to `value`.

Returns: nothing.

Intrinsic function: __rol (TSK51x/TSK52x)

Syntax

```
unsigned char __rol( unsigned char operand, unsigned char count );
```

Use the RL instruction to rotate *operand* left *count* times.

Returns: rotated value.

Example

```
unsigned char c;
int i;

/* rotate left, using int variable */
c = __rol( c, i );
```

Intrinsic function: __ror (TSK51x/TSK52x)

Syntax

```
unsigned char __ror( unsigned char operand, unsigned char count );
```

Use the RR instruction to rotate *operand* right *count* times.

Returns: rotated value.

Example

```
unsigned char c;
int i;

/* rotate right, using constant */
c = __ror( c, 2 );
c = __ror( c, 3 );
c = __ror( c, 7 );
```

Intrinsic function: __setapsr (ARM)

Syntax

```
unsigned int volatile __getapsr( void );
```

Note: This intrinsic is only available for ARMv6-M and ARMv7-M (M-profile architectures).

Set or clear bits in the APSR status register.

Returns: the new value of the APSR status register.

Intrinsic function: __setcpsr (ARM)

Syntax

```
unsigned int volatile __setcpsr( int set, int clear);
```

Set or clear bits in the CPSR status register.

Returns: the new value of the CPSR status register.

Intrinsic function: __setsp (TSK80x)

Syntax

```
void volatile __setsp( unsigned int value );
```

Set the value of the stack pointer SP to `value`.

Returns: nothing.

Intrinsic function: __setspsr (ARM)

Syntax

```
unsigned int volatile __setspsr( int set, int clear);
```

Set or clear bits in the SPSR status register.

Returns: the new value of the SPSR status register.

Example

```
#define SR_F 0x00000040
#define SR_I 0x00000080

i = __setspsr (0, SR_F | SR_I);
if (i & (SR_F | SR_I))
{
  exit (6);    /* Interrupt flags not correct */
  }

if (__getspsr () & (SR_F | SR_I))
{
  exit (7);    /* Interrupt flags not correct */
  }
```

Intrinsic function: __svc (ARM)

Syntax

```
void volatile __svc(int number);
```

Generates a supervisor call (software interrupt). `Number` must be a constant value.

Returns: nothing.

Intrinsic function: __testclear (TSK51x/TSK52x)

Syntax

```
__bit volatile __testclear( __bit *semaphore );
```

Read and clear *semaphore* using the JBC instruction.
Returns: 0 if *semaphore* was not cleared by the JBC instruction, 1 otherwise.

Example

```
__bit b;
unsigned char c;

if ( __testclear( &b ) )              /* JBC instruction */
  c=1;
```

Intrinsic function: __vsp__ (TSK51x/TSK52x)

Syntax

```
__bit __vsp__( void );
```

Virtual stack pointer used. Used in library function va_arg().

Returns: 1 if the virtual stack pointer is used, 0 otherwise.

Standard C Library

The following is a list of all C functions in the *standard* C language.

abort	stops the program
abs	absolute value
acos	arc cosine
asctime	a textual version of the time
asin	arc sine
assert	stops the program if an expression isn't true
atan	arc tangent
atan2	arc tangent, using signs to determine quadrants
atexit	sets a function to be called when the program exits
atof	converts a string to a double
atoi	converts a string to an integer
atol	converts a string to a long
bsearch	perform a binary search
calloc	allocates and clears a two-dimensional chunk of memory
ceil	the smallest integer not less than a certain value
clearerr	clears errors
clock	returns the amount of time that the program has been running
cos	cosine
cosh	hyperbolic cosine
ctime	returns a specifically formatted version of the time
difftime	the difference between two times
div	returns the quotient and remainder of a division
exit	stop the program
exp	returns "e" raised to a given power
fabs	absolute value for floating-point numbers
fclose	close a file
feof	true if at the end-of-file
ferror	checks for a file error
fflush	writes the contents of the output buffer
fgetc	get a character from a stream
fgetpos	get the file position indicator
fgets	get a string of characters from a stream
floor	returns the largest integer not greater than a given value
fmod	returns the remainder of a division
fopen	open a file
fprintf	print formatted output to a file

fputc	write a character to a file
fputs	write a string to a file
fread	read from a file
free	returns previously allocated memory to the operating system
freopen	open an existing stream with a different name
frexp	decomposes a number into scientific notation
fscanf	read formatted input from a file
fseek	move to a specific location in a file
fsetpos	move to a specific location in a file
ftell	returns the current file position indicator
fwrite	write to a file
getc	read a character from a file
getchar	read a character from **STDIN**
getenv	get enviornment information about a variable
gets	read a string from **STDIN**
gmtime	returns a pointer to the current Greenwich Mean Time
isalnum	true if a character is alphanumeric
isalpha	true if a character is alphabetic
iscntrl	true if a character is a control character
isdigit	true if a character is a digit
isgraph	true if a character is a graphical character
islower	true if a character is lowercase
isprint	true if a character is a printing character
ispunct	true if a character is punctuation
isspace	true if a character is a space character
isupper	true if a character is an uppercase character
isxdigit	true if a character is a hexidecimal character
labs	absolute value for long integers
ldexp	computes a number in scientific notation
ldiv	returns the quotient and remainder of a division, in long integer form
localtime	returns a pointer to the current time
log	natural logarithm
log10	natural logarithm, in base 10
longjmp	start execution at a certain point in the program
malloc	allocates memory
memchr	searches an array for the first occurance of a character
memcmp	compares two buffers
memcpy	copies one buffer to another
memmove	moves one buffer to another
memset	fills a buffer with a character

mktime	returns the calendar version of a given time
modf	decomposes a number into integer and fractional parts
perror	displays a string version of the current error to **STDERR**
pow	returns a given number raised to another number
printf	write formatted output to **STDOUT**
putc	write a character to a stream
putchar	write a character to **STDOUT**
puts	write a string to **STDOUT**
qsort	perform a quicksort
raise	send a signal to the program
rand	returns a pseudorandom number
realloc	changes the size of previously allocated memory
remove	erase a file
rename	rename a file
rewind	move the file position indicator to the beginning of a file
scanf	read formatted input from **STDIN**
setbuf	set the buffer for a specific stream
setjmp	set execution to start at a certain point
setlocale	sets the current locale
setvbuf	set the buffer and size for a specific stream
signal	register a function as a signal handler
sin	sine
sinh	hyperbolic sine
sprintf	write formatted output to a buffer
sqrt	square root
srand	initialize the random number generator
sscanf	read formatted input from a buffer
strcat	concatenates two strings
strchr	finds the first occurance of a character in a string
strcmp	compares two strings
strcoll	compares two strings in accordance to the current locale
strcpy	copies one string to another
strcspn	searches one string for any characters in another
strerror	returns a text version of a given error code
strftime	returns individual elements of the date and time
strlen	returns the length of a given string
strncat	concatenates a certain amount of characters of two strings
strncmp	compares a certain amount of characters of two strings
strncpy	copies a certain amount of characters from one string to another
strpbrk	finds the first location of any character in one string, in another string

strrchr	finds the last occurance of a character in a string
strspn	returns the length of a substring of characters of a string
strstr	finds the first occurance of a substring of characters
strtod	converts a string to a double
strtok	finds the next token in a string
strtol	converts a string to a long
strtoul	converts a string to an unsigned long
strxfrm	converts a substring so that it can be used by string comparison functions
system	perform a system call
tan	tangent
tanh	hyperbolic tangent
time	returns the current calendar time of the system
tmpfile	return a pointer to a temporary file
tmpnam	return a unique filename
tolower	converts a character to lowercase
toupper	converts a character to uppercase
ungetc	puts a character back into a stream
va_arg	use variable length parameter lists
vprintf, vfprintf, and vsprintf	write formatted output with variable argument lists

Standard C Date & Time Functions

The following is a list of all Standard C Date & Time functions.

asctime	a textual version of the time
clock	returns the amount of time that the program has been running
ctime	returns a specifically formatted version of the time
difftime	the difference between two times
gmtime	returns a pointer to the current Greenwich Mean Time
localtime	returns a pointer to the current time
mktime	returns the calendar version of a given time
setlocale	sets the current locale
strftime	returns individual elements of the date and time
time	returns the current calendar time of the system

Standard C date & time function: asctime

Syntax

```
#include <time.h>
char *asctime( const struct tm *ptr );
```

The function asctime() converts the time in the struct 'ptr' to a character string of the following format:

```
day month date hours:minutes:seconds year
```

An example:

```
Mon Jun 26 12:03:53 2000
```

Standard C date & time function: clock

Syntax

```
#include <time.h>
clock_t clock( void );
```

The clock() function returns the processor time since the program started, or -1 if that information is unavailable. To convert the return value to seconds, divide it by CLOCKS_PER_SEC. (Note: if your compiler is POSIX compliant, then CLOCKS_PER_SEC is always defined as 1000000.)

Standard C date & time function: ctime

Syntax

```
#include <time.h>
char *ctime( const time_t *time );
```

The ctime() function converts the calendar time time to local time of the format:

```
day month date hours:minutes:seconds year
```

using ctime() is equivalent to

```
asctime( localtime( tp ) );
```

Standard C date & time function: difftime

Syntax

```
#include <time.h>
double difftime( time_t time2, time_t time1 );
```

The function difftime() returns `time2` - `time1`, in seconds.

Standard C date & time function: gmtime

Syntax

```
#include <time.h>
struct tm *gmtime( const time_t *time );
```

The gmtime() function returns the given *time* in Coordinated Universal Time (usually Greenwich mean time), unless it's not supported by the system, in which case **NULL** is returned. Watch out for static return.

Standard C date & time function: localtime

Syntax

```
#include <time.h>
struct tm *localtime( const time_t *time );
```

The function localtime() converts calendar time time into local time. Watch out for the static return.

Standard C date & time function: mktime

Syntax

```
#include <time.h>
time_t mktime( struct tm *time );
```

The mktime() function converts the local time in *time* to calendar time, and returns it. If there is an error, -1 is returned.

Standard C date & time function: setlocale

Syntax

```
#include <locale.h>
char *setlocale( int category, const char * locale );
```

The setlocale() function is used to set and retrieve the current locale. If *locale* is **NULL**, the current locale is returned. Otherwise, *locale* is used to set the locale for the given *category*.
category can have the following values:

Value	Description
LC_ALL	All of the locale
LC_TIME	Date and time formatting
LC_NUMERIC	Number formatting
LC_COLLATE	String collation and regular expression matching
LC_CTYPE	Regular expression matching, conversion, case-sensitive comparison, wide character functions, and character classification.
LC_MONETARY	For monetary formatting
LC_MESSAGES	For natural language messages

Standard C date & time function: strftime

Syntax

```
#include <time.h>
size_t strftime( char *str, size_t maxsize, const char *fmt, struct tm *time );
```

The function strftime() formats date and time information from *time* to a format specified by *fmt*, then stores the result in *str* (up to *maxsize* characters). Certain codes may be used in *fmt* to specify different types of time:

Code	Meaning
%a	abbreviated weekday name (e.g. Fri)
%A	full weekday name (e.g. Friday)
%b	abbreviated month name (e.g. Oct)
%B	full month name (e.g. October)
%c	the standard date and time string
%d	day of the month, as a number (1-31)
%H	hour, 24 hour format (0-23)
%I	hour, 12 hour format (1-12)
%j	day of the year, as a number (1-366)
%m	month as a number (1-12).
%M	minute as a number (0-59)
%p	locale's equivalent of AM or PM
%S	second as a number (0-59)
%U	week of the year, (0-53), where week 1 has the first Sunday
%w	weekday as a decimal (0-6), where Sunday is 0
%W	week of the year, (0-53), where week 1 has the first Monday
%x	standard date string
%X	standard time string
%y	year in decimal, without the century (0-99)
%Y	year in decimal, with the century
%Z	time zone name
%%	a percent sign

The strftime() function returns the number of characters put into *str*, or zero if an error occurs.

Standard C date & time function: time

Syntax

```
#include <time.h>
time_t time( time_t *time );
```

The function time() returns the current time, or -1 if there is an error. If the argument 'time' is given, then the current time is stored in 'time'.

Standard C I/O Functions

The following is a list of all Standard C I/O functions. For C++, these functions provide an alternative to the C++ stream-based I/O classes.

clearerr	clears errors
fclose	close a file
feof	true if at the end-of-file
ferror	checks for a file error
fflush	writes the contents of the output buffer
fgetc	get a character from a stream
fgetpos	get the file position indicator
fgets	get a string of characters from a stream
fopen	open a file
fprintf	print formatted output to a file
fputc	write a character to a file
fputs	write a string to a file
fread	read from a file
freopen	open an existing stream with a different name
fscanf	read formatted input from a file
fseek	move to a specific location in a file
fsetpos	move to a specific location in a file
ftell	returns the current file position indicator
fwrite	write to a file
getc	read a character from a file
getchar	read a character from **stdin**
gets	read a string from **stdin**
perror	displays a string version of the current error to **stderr**
printf	write formatted output to **stdout**
putc	write a character to a stream
putchar	write a character to **stdout**
puts	write a string to **stdout**
remove	erase a file
rename	rename a file
rewind	move the file position indicator to the beginning of a file
scanf	read formatted input from **stdin**
setbuf	set the buffer for a specific stream
setvbuf	set the buffer and size for a specific stream
sprintf	write formatted output to a buffer
sscanf	read formatted input from a buffer
tmpfile	return a pointer to a temporary file

tmpnam	return a unique filename
ungetc	puts a character back into a stream
vprintf, vfprintf, and vsprintf	write formatted output with variable argument lists

Standard C I/O function: clearerr

Syntax

```
#include <stdio.h>
void clearerr( FILE *stream );
```

The clearerr function resets the error flags and **EOF** indicator for the given *`stream`*. When an error occurs, you can use perror() to figure out which error actually occurred.

Standard C I/O function: fclose

Syntax

```
#include <stdio.h>
int fclose( FILE *stream );
```

The function fclose() closes the given file stream, deallocating any buffers associated with that stream. fclose() returns 0 upon success, and **EOF** otherwise.

Standard C I/O function: feof

Syntax

```
#include <stdio.h>
int feof( FILE *stream );
```

The function feof() returns a nonzero value if the end of the given file *`stream`* has been reached.

Standard C I/O function: ferror

Syntax

```
#include <stdio.h>
int ferror( FILE *stream );
```

The ferror() function looks for errors with *`stream`*, returning zero if no errors have occured, and non-zero if there is an error. In case of an error, use perror() to determine which error has occured.

Standard C I/O function: fflush

Syntax

```
#include <stdio.h>
int fflush( FILE *stream );
```

If the given file *stream* is an output stream, then fflush() causes the output buffer to be written to the file. If the given *stream* is of the input type, then fflush() causes the input buffer to be cleared. fflush() is useful when debugging, if a program segfaults before it has a chance to write output to the screen. Calling fflush(**stdout**) directly after debugging output will ensure that your output is displayed at the correct time.

```
printf( "Before first call\n" );
fflush( stdout );
shady_function();
printf( "Before second call\n" );
fflush( stdout );
dangerous_dereference();
```

Standard C I/O function: fgetc

Syntax

```
#include <stdio.h>
int fgetc( FILE *stream );
```

The fgetc() function returns the next character from *stream*, or **EOF** if the end of file is reached or if there is an error.

Standard C I/O function: fgetpos

Syntax

```
#include <stdio.h>
int fgetpos( FILE *stream, fpos_t *position );
```

The fgetpos() function stores the file position indicator of the given file *stream* in the given *position* variable. The position variable is of type fpos_t (which is defined in stdio.h) and is an object that can hold every possible position in a FILE. fgetpos() returns zero upon success, and a non-zero value upon failure.

Standard C I/O function: fgets

Syntax

```
#include <stdio.h>
char *fgets( char *str, int num, FILE *stream );
```

The function fgets() reads up to *num* - 1 characters from the given file *stream* and dumps them into *str*. The string that fgets() produces is always **NULL**-terminated. fgets() will stop when it reaches the end of a line, in which case *str* will contain that newline character. Otherwise, fgets() will stop when it reaches *num* - 1 characters or encounters the **EOF** character. fgets() returns *str* on success, and **NULL** on an error.

Standard C I/O function: fopen

Syntax

```
#include <stdio.h>
FILE *fopen( const char *fname, const char *mode );
```

The fopen() function opens a file indicated by *fname* and returns a stream associated with that file. If there is an error, fopen() returns **NULL**. *mode* is used to determine how the file will be treated (i.e. for input, output, etc)

Mode	Meaning
"r"	Open a text file for reading
"w"	Create a text file for writing
"a"	Append to a text file
"rb"	Open a binary file for reading
"wb"	Create a binary file for writing
"ab"	Append to a binary file
"r+"	Open a text file for read/write
"w+"	Create a text file for read/write
"a+"	Open a text file for read/write
"rb+"	Open a binary file for read/write
"wb+"	Create a binary file for read/write
"ab+"	Open a binary file for read/write

An example:

```
int ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
```

Standard C I/O function: fprintf

Syntax

```
#include <stdio.h>
int fprintf( FILE *stream, const char *format, ... );
```

The fprintf() function sends information (the arguments) according to the specified *format* to the file indicated by *stream*. fprintf() works just like printf() as far as the format goes. The return value of fprintf() is the number of characters outputted, or a negative number if an error occurs. An example:

```
char name[20] = "Mary";
FILE *out;
out = fopen( "output.txt", "w" );
if( out != NULL )
  fprintf( out, "Hello %s\n", name );
```

Standard C I/O function: fputc

Syntax

```
#include <stdio.h>
int fputc( int ch, FILE *stream );
```

The function fputc() writes the given character *ch* to the given output *stream*. The return value is the character, unless there is an error, in which case the return value is **EOF**.

Standard C I/O function: fputs

Syntax

```
#include <stdio.h>
int fputs( const char *str, FILE *stream );
```

The fputs() function writes an array of characters pointed to by *str* to the given output *stream*. The return value is non-negative on success, and **EOF** on failure.

Standard C I/O function: fread

Syntax

```
#include <stdio.h>
int fread( void *buffer, size_t size, size_t num, FILE *stream );
```

The function fread() reads *num* number of objects (where each object is *size* bytes) and places them into the array pointed to by buffer. The data comes from the given input *stream*. The return value of the function is the number of things read. You can use feof() or ferror() to figure out if an error occurs.

Standard C I/O function: freopen

Syntax

```
#include <stdio.h>
FILE *freopen( const char *fname, const char *mode, FILE *stream );
```

The freopen() function is used to reassign an existing *stream* to a different file and mode. After a call to this function, the given file *stream* will refer to *fname* with access given by *mode*. The return value of freopen() is the new stream, or **NULL** if there is an error.

Standard C I/O function: fscanf

Syntax

```
#include <stdio.h>
int fscanf( FILE *stream, const char *format, ... );
```

The function fscanf() reads data from the given file *stream* in a manner exactly like scanf(). The return value of fscanf() is the number of variables that are actually assigned values, or **EOF** if no assignments could be made.

Standard C I/O function: fseek

Syntax

```
#include <stdio.h>
int fseek( FILE *stream, long offset, int origin );
```

The function fseek() sets the file position data for the given *stream*. The origin value should have one of the following values (defined in stdio.h):

Name	Explanation
SEEK_SET	Seek from the start of the file
SEEK_CUR	Seek from the current location
SEEK_END	Seek from the end of the file

fseek() returns zero upon success, non-zero on failure. You can use fseek() to move beyond a file, but not before the beginning. Using fseek() clears the **EOF** flag associated with that stream.

Standard C I/O function: fsetpos

Syntax

```
#include <stdio.h>
int fsetpos( FILE *stream, const fpos_t *position );
```

The fsetpos() function moves the file position indicator for the given *stream* to a location specified by the *position* object. fpos_t is defined in stdio.h. The return value for fsetpos() is zero upon success, non-zero on failure.

Standard C I/O function: ftell

Syntax

```
#include <stdio.h>
long ftell( FILE *stream );
```

The ftell() function returns the current file position for *stream*, or -1 if an error occurs.

Standard C I/O function: fwrite

Syntax

```
#include <stdio.h>
int fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
```

The fwrite() function writes, from the array *buffer*, *count* objects of size *size* to *stream*. The return value is the number of objects written.

Standard C I/O function: getc

Syntax

```
#include <stdio.h>
int getc( FILE *stream );
```

The getc() function returns the next character from *stream*, or **EOF** if the end of file is reached. getc() is identical to fgetc(). For example:

```
int ch;
FILE *input = fopen( "stuff", "r" );

ch = getc( input );
while( ch != EOF )
{
  printf( "%c", ch );
  ch = getc( input );
}
```

Standard C I/O function: getchar

Syntax

```
#include <stdio.h>
int getchar( void );
```

The getchar() function returns the next character from **stdin**, or **EOF** if the end of file is reached.

Standard C I/O function: gets

Syntax

```
#include <stdio.h>
char *gets( char *str );
```

The gets() function reads characters from **stdin** and loads them into *str*, until a newline or **EOF** is reached. The newline character is translated into a null termination. The return value of gets() is the read-in string, or **NULL** if there is an error. Note that gets() does not perform bounds checking, and thus risks overrunning *str*. For a similar (and safer) function that includes bounds checking, see fgets().

Standard C I/O function: perror

Syntax

```
#include <stdio.h>
void perror( const char *str );
```

The perror() function prints `str` and an implementation-defined error message corresponding to the global variable `errno`. For example:

```
char* input_filename = "not_found.txt";
FILE* input = fopen( input_filename, "r" );
if( input == NULL )
{
  char error_msg[255];
  sprintf( error_msg, "Error opening file '%s'", input_filename );
  perror( error_msg );
  exit( -1 );
}
```

The the file called `not_found.txt` is not found, this code will produce the following output:

```
Error opening file 'not_found.txt': No such file or directory
```

Standard C I/O function: printf

Syntax

```
#include <stdio.h>
int printf( const char *format, ... );
```

The printf() function prints output to **stdout**, according to `format` and other arguments passed to printf(). The string `format` consists of two types of items - characters that will be printed to the screen, and format commands that define how the other arguments to printf() are displayed. Basically, you specify a format string that has text in it, as well as "special" characters that map to the other arguments of printf(). For example, this code

```
char name[20] = "Bob";
int age = 21;
printf( "Hello %s, you are %d years old\n", name, age );
```

displays the following output:

```
Hello Bob, you are 21 years old
```

The %s means, "insert the first argument, a string, right here." The %d indicates that the second argument (an integer) should be placed there. There are different %-codes for different variable types, as well as options to limit the length of the variables.

Code	Format
%c	character
%d	signed integers
%i	signed integers
%e	scientific notation, with a lowercase "e"
%E	scientific notation, with a uppercase "E"
%f	floating point

%g	use %e or %f, whichever is shorter
%G	use %E or %f, whichever is shorter
%o	octal
%s	a string of characters
%u	unsigned integer
%x	unsigned hexadecimal, with lowercase letters
%X	unsigned hexadecimal, with uppercase letters
%p	a pointer
%n	the argument shall be a pointer to an integer into which is placed the number of characters written so far
%%	a '%' sign

An integer placed between a `%` sign and the format command acts as a minimum field width specifier, and pads the output with spaces or zeros to make it long enough. If you want to pad with zeros, place a zero before the minimum field width specifier:

```
%012d
```

You can also include a precision modifier, in the form of a `.N` where *N* is some number, before the format command:

```
%012.4d
```

The precision modifier has different meanings depending on the format command being used:

- With `%e`, `%E`, and `%f`, the precision modifier lets you specify the number of decimal places desired. For example, `%12.6f` will display a floating number at least 12 digits wide, with six decimal places.
- With `%g` and `%G`, the precision modifier determines the maximum number of significant digits displayed.
- With `%s`, the precision modifer simply acts as a maximumfield length, to complement the minimum field length that precedes the period.

All of printf()'s output is right-justified, unless you place a minus sign right after the `%` sign. For example,

```
%-12.4f
```

will display a floating point number with a minimum of 12 characters, 4 decimal places, and left justified. You may modify the `%d`, `%i`, `%o`, `%u`, and `%x` type specifiers with the letter `l` (el) and the letter `h` to specify long and short data types (e.g. `%hd` means a short integer). The `%e`, `%f`, and `%g` type specifiers can have the letter `l` before them to indicate that a double follows. The `%g`, `%f`, and `%e` type specifiers can be preceded with the character '#' to ensure that the decimal point will be present, even if there are no decimal digits. The use of the '#' character with the `%x` type specifier indicates that the hexidecimal number should be printed with the '0x' prefix. The use of the '#' character with the `%o` type specifier indicates that the octal value should be displayed with a `0` prefix.

Inserting a plus sign '+' into the type specifier will force positive values to be preceded by a '+' sign. Putting a space character `' '` there will force positive values to be preceded by a single space character.

You can also include constant escape sequences in the output string.

The return value of printf() is the number of characters printed, or a negative number if an error occurred.

Standard C I/O function: putc

Syntax

```
#include <stdio.h>
int putc( int ch, FILE *stream );
```

The putc() function writes the character *ch* to *stream*. The return value is the character written, or **EOF** if there is an error. For example:

```
int ch;
FILE *input, *output;
input = fopen( "tmp.c", "r" );
output = fopen( "tmpCopy.c", "w" );
ch = getc( input );
while( ch != EOF )
{
  putc( ch, output );
  ch = getc( input );
}
fclose( input );
fclose( output );
```

generates a copy of the file *tmp.c* called *tmpCopy.c*.

Standard C I/O function: putchar

Syntax

```
#include <stdio.h>
int putchar( int ch );
```

The putchar() function writes *ch* to **stdout**. The code

```
putchar( ch );
```

is the same as

```
putc( ch, stdout );
```

The return value of putchar() is the written character, or **EOF** if there is an error.

Standard C I/O function: puts

Syntax

```
#include <stdio.h>
int puts( char *str );
```

The function puts() writes *str* to **stdout**. puts() returns non-negative on success, or **EOF** on failure.

Standard C I/O function: remove

Syntax

```
#include <stdio.h>
int remove( const char *fname );
```

The remove() function erases the file specified by `fname`. The return value of remove() is zero upon success, and non-zero if there is an error.

Standard C I/O function: rename

Syntax

```
#include <stdio.h>
int rename( const char *oldfname, const char *newfname );
```

The function rename() changes the name of the file `oldfname` to `newfname`. The return value of rename() is zero upon success, non-zero on error.

Standard C I/O function: rewind

Syntax

```
#include <stdio.h>
void rewind( FILE *stream );
```

The function rewind() moves the file position indicator to the beginning of the specified `stream`, also clearing the error and **EOF** flags associated with that stream.

Standard C I/O function: scanf

Syntax

```
#include <stdio.h>
int scanf( const char *format, ... );
```

The scanf() function reads input from **stdin**, according to the given `format`, and stores the data in the other arguments. It works a lot like printf(). The `format` string consists of control characters, whitespace characters, and non-whitespace characters. The control characters are preceded by a % sign, and are as follows:

Control Character	Explanation
%c	a single character
%d	a decimal integer
%i	an integer
%e, %f, %g	a floating-point number
%lf	a double
%o	an octal number
%s	a string
%x	a hexadecimal number
%p	a pointer
%n	an integer equal to the number of characters read so far
%u	an unsigned integer
%[ ]	a set of characters
%%	a percent sign

scanf() reads the input, matching the characters from format. When a control character is read, it puts the value in the next variable. Whitespace (tabs, spaces, etc) are skipped. Non-whitespace characters are matched to the input, then discarded. If a number comes between the `%` sign and the control character, then only that many characters will be converted into the variable. If scanf() encounters a set of characters, denoted by the `%[]` control character, then any characters found within the brackets are read into the variable. The return value of scanf() is the number of variables that were successfully assigned values, or **EOF** if there is an error.

Example

This code snippet uses scanf() to read an int, float, and a double from the user. Note that the variable arguments to scanf() are passed in by address, as denoted by the ampersand `(&)` preceding each variable:

```
int i;
float f;
double d;

printf( "Enter an integer: " );
scanf( "%d", &i );

printf( "Enter a float: " );
scanf( "%f", &f );

printf( "Enter a double: " );
scanf( "%lf", &d );

printf( "You entered %d, %f, and %f\n", i, f, d );
```

Standard C I/O function: setbuf

Syntax

```
#include <stdio.h>
void setbuf( FILE *stream, char *buffer );
```

The setbuf() function sets *`stream`* to use *`buffer`*, or, if *`buffer`* is null, turns off buffering. If a non-standard buffer size is used, it should be BUFSIZ characters long.

Standard C I/O function: setvbuf

Syntax

```
#include <stdio.h>
int setvbuf( FILE *stream, char *buffer, int mode, size_t size );
```

The function setvbuf() sets the buffer for *`stream`* to be *`buffer`*, with a size of *`size`*. *`mode`* can be:

- _IOFBF, which indicates full buffering
- _IOLBF, which means line buffering
- _IONBF, which means no buffering

Standard C I/O function: sprintf

Syntax

```
#include <stdio.h>
int sprintf( char *buffer, const char *format, ... );
```

The sprintf() function is just like printf(), except that the output is sent to *`buffer`*. The return value is the number of characters written. For example:

```
char string[50];
int file_number = 0;

sprintf( string, "file.%d", file_number );
file_number++;
output_file = fopen( string, "w" );
```

Note that sprintf() does the opposite of a function like atoi() -- where atoi() converts a string into a number, sprintf() can be used to convert a number into a string.

For example, the following code uses sprintf() to convert an integer into a string of characters:

```
char result[100];
int num = 24;
sprintf( result, "%d", num );
```

This code is similar, except that it converts a floating-point number into an array of characters:

```
char result[100];
float fnum = 3.14159;
sprintf( result, "%f", fnum );
```

Standard C I/O function: sscanf

Syntax

```
#include <stdio.h>
int sscanf( const char *buffer, const char *format, ... );
```

The function sscanf() is just like scanf(), except that the input is read from *`buffer`*.

Standard C I/O function: tmpfile

Syntax

```
#include <stdio.h>
FILE *tmpfile( void );
```

The function tmpfile() opens a temporary file with an unique filename and returns a pointer to that file. If there is an error, null is returned.

Standard C I/O function: tmpnam

Syntax

```
#include <stdio.h>
char *tmpnam( char *name );
```

The tmpnam() function creates an unique filename and stores it in *name*. tmpnam() can be called up to **TMP_MAX** times.

Standard C I/O function: ungetc

Syntax

```
#include <stdio.h>
int ungetc( int ch, FILE *stream );
```

The function ungetc() puts the character *ch* back in *stream*.

Standard C I/O function: vprintf, vfprintf, and vsprintf

Syntax

```
#include <stdarg.h>
#include <stdio.h>
int vprintf( char *format, va_list arg_ptr );
int vfprintf( FILE *stream, const char *format, va_list arg_ptr );
int vsprintf( char *buffer, char *format, va_list arg_ptr );
```

These functions are very much like printf(), fprintf(), and sprintf(). The difference is that the argument list is a pointer to a list of arguments. **va_list** is defined in stdarg.h, and is also used by (Other Standard C Functions) va_arg(). For example:

```
void error( char *fmt, ... )
{
  va_list args;
  va_start( args, fmt );
  fprintf( stderr, "Error: " );
  vfprintf( stderr, fmt, args );
  fprintf( stderr, "\n" );
  va_end( args );
  exit( 1 );
}
```

Standard C Math Functions

The following is a list of all Standard C Math functions.

abs	absolute value
acos	arc cosine
asin	arc sine
atan	arc tangent
atan2	arc tangent, using signs to determine quadrants
ceil	the smallest integer not less than a certain value
cos	cosine
cosh	hyperbolic cosine
div	returns the quotient and remainder of a division
exp	returns "e" raised to a given power
fabs	absolute value for floating-point numbers
floor	returns the largest integer not greater than a given value
fmod	returns the remainder of a division
frexp	decomposes a number into scientific notation
labs	absolute value for long integers
ldexp	computes a number in scientific notation
ldiv	returns the quotient and remainder of a division, in long integer form
log	natural logarithm (to base e)
log10	common logarithm (to base 10)
modf	decomposes a number into integer and fractional parts
pow	returns a given number raised to another number
sin	sine
sinh	hyperbolic sine
sqrt	square root
tan	tangent
tanh	hyperbolic tangent

Standard C math function: abs

Syntax

```
#include <stdlib.h>
int abs( int num );
```

The abs() function returns the absolute value of *num*. For example:

```
int magic_number = 10;
printf("Enter a guess: ");
scanf("%d", &x);
printf("Your guess was %d away from the magic number.\n", abs(magic_number - x));
```

Standard C math function: acos

Syntax

```
#include <math.h>
double acos( double arg );
```

The acos() function returns the arc cosine of *arg*, which will be in the range [0, pi]. *arg* should be between -1 and 1. If *arg* is outside this range, acos() returns NAN and raises a floating-point exception.

Standard C math function: asin

Syntax

```
#include <math.h>
double asin( double arg );
```

The asin() function returns the arc sine of *arg*, which will be in the range [-pi/2, +pi/2]. *arg* should be between -1 and 1. If *arg* is outside this range, asin() returns NAN and raises a floating-point exception.

Standard C math function: atan

Syntax

```
#include <math.h>
double atan( double arg );
```

The function atan() returns the arc tangent of *arg*, which will be in the range [-pi/2, +pi/2].

Standard C math function: atan2

Syntax

```
#include <math.h>
double atan2( double y, double x );
```

The atan2() function computes the arc tangent of *y*/*x*, using the signs of the arguments to compute the quadrant of the return value.

Note the order of the arguments passed to this function.

Standard C math function: ceil

Syntax

```
#include <math.h>
double ceil( double num );
```

The ceil() function returns the smallest integer no less than *num*. For example,

```
y = 6.04;
x = ceil( y );
```

would set x to 7.0.

Standard C math function: cos

Syntax

```
#include <math.h>
double cos( double arg );
```

The cos() function returns the cosine of *arg*, where *arg* is expressed in radians. The return value of cos() is in the range [-1,1]. If *arg* is infinite, cos() will return NAN and raise a floating-point exception.

Standard C math function: cosh

Syntax

```
#include <math.h>
double cosh( double arg );
```

The function cosh() returns the hyperbolic cosine of *arg*.

Standard C math function: div

Syntax

```
#include <stdlib.h>
div_t div( int numerator, int denominator );
```

The function div() returns the quotient and remainder of the operation *numerator* / *denominator*. The **div_t** structure is defined in stdlib.h, and has at least:

```
int quot;  // The quotient
int rem;    // The remainder
```

For example, the following code displays the quotient and remainder of x/y:

```
div_t temp;
temp = div( x, y );
printf( "%d divided by %d yields %d with a remainder of %d\n",
        x, y, temp.quot, temp.rem );
```

Standard C math function: exp

Syntax

```
#include <math.h>
double exp( double arg );
```

The exp() function returns e (2.7182818) raised to the *arg*th power.

Standard C math function: fabs

Syntax

```
#include <math.h>
double fabs( double arg );
```

The function fabs() returns the absolute value of *arg*.

Standard C math function: floor

Syntax

```
#include <math.h>
double floor( double arg );
```

The function floor() returns the largest integer not greater than *arg*. For example,

```
y = 6.04;
x = floor( y );
```

would result in x being set to 6.0.

Standard C math function: fmod

Syntax

```
#include <math.h>
double fmod( double x, double y );
```

The fmod() function returns the remainder of *x*/*y*.

Standard C math function: frexp

Syntax

```
#include <math.h>
double frexp( double num, int* exp );
```

The function frexp() is used to decompose *num* into two parts: a mantissa between 0.5 and 1 (returned by the function) and an exponent returned as *exp*. Scientific notation works like this:

```
num = mantissa * (2 ^ exp)
```

Standard C math function: labs

Syntax

```
#include <stdlib.h>
long labs( long num );
```

The function labs() returns the absolute value of *num*.

Standard C math function: ldexp

Syntax

```
#include <math.h>
double ldexp( double num, int exp );
```

The ldexp() function returns *num* * (2 ^ *exp*). And get this: if an overflow occurs, **HUGE_VAL** is returned.

Standard C math function: ldiv

Syntax

```
#include <stdlib.h>
ldiv_t ldiv( long numerator, long denominator );
```

Testing: **a**div_t, **div_t**, **l**div_t.

The ldiv() function returns the quotient and remainder of the operation *numerator* / *denominator*. The **l**div_t structure is defined in stdlib.h and has at least:

```
long quot;  // the quotient
long rem;  // the remainder
```

Standard C math function: log

Syntax

```
#include <math.h>
double log( double num );
```

The function log() returns the natural (base e) logarithm of *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

In order to calculate the logarithm of *x* to an arbitrary base *b*, you can use:

```
double answer = log(x) / log(b);
```

Standard C math function: log10

Syntax

```
#include <math.h>
double log10( double num );
```

The log10() function returns the base 10 (or common) logarithm for *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

Standard C math function: modf

Syntax

```
#include <math.h>
double modf( double num, double *i );
```

The function modf() splits *num* into its integer and fraction parts. It returns the fractional part and loads the integer part into *i*.

Standard C math function: pow

Syntax

```
#include <math.h>
double pow( double base, double exp );
```

The pow() function returns *base* raised to the *exp*th power. There's a domain error if *base* is zero and *exp* is less than or equal to zero. There's also a domain error if *base* is negative and *exp* is not an integer. There's a range error if an overflow occurs.

Standard C math function: sin

Syntax

```
#include <math.h>
double sin( double arg );
```

The function sin() returns the sine of *arg*, where *arg* is given in radians. The return value of sin() will be in the range [-1,1]. If *arg* is infinite, sin() will return NAN and raise a floating-point exception.

Standard C math function: sinh

Syntax

```
#include <math.h>
double sinh( double arg );
```

The function sinh() returns the hyperbolic sine of *arg*.

Standard C math function: sqrt

Syntax

```
#include <math.h>
double sqrt( double num );
```

The sqrt() function returns the square root of *num*. If *num* is negative, a domain error occurs.

Standard C math function: tan

Syntax

```
#include <math.h>
double tan( double arg );
```

The tan() function returns the tangent of *arg*, where *arg* is given in radians. If *arg* is infinite, tan() will return NAN and raise a floating-point exception.

Standard C math function: tanh

Syntax

```
#include <math.h>
double tanh( double arg );
```

The function tanh() returns the hyperbolic tangent of *arg*.

Standard C Memory Functions

The following is a list of all Standard C Memory functions.

calloc	allocates and clears a two-dimensional chunk of memory
free	returns previously allocated memory to the operating system
malloc	allocates memory
realloc	changes the size of previously allocated memory

Standard C memory function: calloc

Syntax

```
#include <stdlib.h>
void* calloc( size_t num, size_t size );
```

The calloc() function returns a pointer to space for an array of *num* objects, each of size *size*. The newly allocated memory is initialized to zero.

calloc() returns **NULL** if there is an error.

Standard C memory function: free

Syntax

```
#include <stdlib.h>
void free( void* ptr );
```

The free() function deallocates the space pointed to by *ptr*, freeing it up for future use. *ptr* must have been used in a previous call to malloc(), calloc(), or realloc(). An example:

```
typedef struct data_type
{
  int age;
  char name[20];
} data;

data *willy;
willy = (data*) malloc( sizeof(*willy) );
...
free( willy );
```

Standard C memory function: malloc

Syntax

```
#include <stdlib.h>
void *malloc( size_t size );
```

The function malloc() returns a pointer to a chunk of memory of size *size*, or **NULL** if there is an error. The memory pointed to will be on the heap, not the stack, so make sure to free it when you are done with it. An example:

```
typedef struct data_type
{
  int age;
  char name[20];
} data;

data *bob;
bob = (data*) malloc( sizeof(data) );
if( bob != NULL )
{
  bob->age = 22;
  strcpy( bob->name, "Robert" );
  printf( "%s is %d years old\n", bob->name, bob->age );
}
free( bob );
```

Standard C memory function: realloc

Syntax

```
#include <stdlib.h>
void *realloc( void *ptr, size_t size );
```

The realloc() function changes the size of the object pointed to by ptr to the given size. size can be any size, larger or smaller than the original. The return value is a pointer to the new space, or **NULL** if there is an error.

Standard C String and Character Functions

The following is a list of all Standard C String and Character functions.

atof	converts a string to a double
atoi	converts a string to an integer
atol	converts a string to a long
isalnum	true if a character is alphanumeric
isalpha	true if a character is alphabetic
iscntrl	true if a character is a control character
isdigit	true if a character is a digit
isgraph	true if a character is a graphical character
islower	true if a character is lowercase
isprint	true if a character is a printing character
ispunct	true if a character is punctuation
isspace	true if a character is a space character
isupper	true if a character is an uppercase character
isxdigit	true if a character is a hexidecimal character
memchr	searches an array for the first occurance of a character
memcmp	compares two buffers
memcpy	copies one buffer to another
memmove	moves one buffer to another
memset	fills a buffer with a character
strcat	concatenates two strings
strchr	finds the first occurance of a character in a string
strcmp	compares two strings
strcoll	compares two strings in accordance to the current locale
strcpy	copies one string to another
strcspn	searches one string for any characters in another
strerror	returns a text version of a given error code
strlen	returns the length of a given string
strncat	concatenates a certain amount of characters of two strings
strncmp	compares a certain amount of characters of two strings
strncpy	copies a certain amount of characters from one string to another
strpbrk	finds the first location of any character in one string, in another string
strrchr	finds the last occurance of a character in a string
strspn	returns the length of a substring of characters of a string
strstr	finds the first occurance of a substring of characters
strtod	converts a string to a double
strtok	finds the next token in a string

strtol	converts a string to a long
strtoul	converts a string to an unsigned long
strxfrm	converts a substring so that it can be used by string comparison functions
tolower	converts a character to lowercase
toupper	converts a character to uppercase

Standard C string and character function: atof

Syntax

```
#include <stdlib.h>
double atof( const char *str );
```

The function atof() converts *str* into a double, then returns that value. *str* must start with a valid number, but can be terminated with any non-numerical character, other than "E" or "e". For example,

```
x = atof( "42.0is_the_answer" );
```

results in x being set to 42.0.

Standard C string and character function: atoi

Syntax

```
#include <stdlib.h>
int atoi( const char *str );
```

The atoi() function converts *str* into an integer, and returns that integer. *str* should start with whitespace or some sort of number, and atoi() will stop reading from *str* as soon as a non-numerical character has been read. For example:

```
int i;
i = atoi( "512" );
i = atoi( "512.035" );
i = atoi( "  512.035" );
i = atoi( "  512+34" );
i = atoi( "  512 bottles of beer on the wall" );
```

All five of the above assignments to the variable *i* would result in it being set to 512.

If the conversion cannot be performed, then atoi() will return zero:

```
int i = atoi( " does not work: 512" );  // results in i == 0
```

You can use sprintf() to convert a number into a string.

Standard C string and character function: atol

Syntax

```
#include <stdlib.h>
long atol( const char *str );
```

The function atol() converts `str` into a long, then returns that value. atol() will read from `str` until it finds any character that should not be in a long. The resulting truncated value is then converted and returned. For example,

```
x = atol( "1024.0001" );
```

results in x being set to 1024L.

Standard C string and character function: isalnum

Syntax

```
#include <ctype.h>
int isalnum( int ch );
```

The function isalnum() returns non-zero if its argument is a numeric digit or a letter of the alphabet. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isalnum(c) )
  printf( "You entered the alphanumeric character %c\n", c );
```

Standard C string and character function: isalpha

Syntax

```
#include <ctype.h>
int isalpha( int ch );
```

The function isalpha() returns non-zero if its argument is a letter of the alphabet. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isalpha(c) )
  printf( "You entered a letter of the alphabet\n" );
```

Standard C string and character function: iscntrl

Syntax

```
#include <ctype.h>
int iscntrl( int ch );
```

The iscntrl() function returns non-zero if its argument is a control character (between 0 and 0x1F or equal to 0x7F). Otherwise, zero is returned.

Standard C string and character function: isdigit

Syntax

```
#include <ctype.h>
int isdigit( int ch );
```

The function isdigit() returns non-zero if its argument is a digit between 0 and 9. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isdigit(c) )
  printf( "You entered the digit %c\n", c );
```

Standard C string and character function: isgraph

Syntax

```
#include <ctype.h>
int isgraph( int ch );
```

The function isgraph() returns non-zero if its argument is any printable character other than a space (if you can see the character, then isgraph() will return a non-zero value). Otherwise, zero is returned.

Standard C string and character function: islower

Syntax

```
#include <ctype.h>
int islower( int ch );
```

The islower() function returns non-zero if its argument is a lowercase letter. Otherwise, zero is returned.

Standard C string and character function: isprint

Syntax

```
#include <ctype.h>
int isprint( int ch );
```

The function isprint() returns non-zero if its argument is a printable character (including a space). Otherwise, zero is returned.

Standard C string and character function: ispunct

Syntax

```
#include <ctype.h>
int ispunct( int ch );
```

The ispunct() function returns non-zero if its argument is a printing character but neither alphanumeric nor a space. Otherwise, zero is returned.

Standard C string and character function: isspace

Syntax

```
#include <ctype.h>
int isspace( int ch );
```

The isspace() function returns non-zero if its argument is some sort of space (i.e. single space, tab, vertical tab, form feed, carriage return, or newline). Otherwise, zero is returned.

Standard C string and character function: isupper

Syntax

```
#include <ctype.h>
int isupper( int ch );
```

The isupper() function returns non-zero if its argument is an uppercase letter. Otherwise, zero is returned.

Standard C string and character function: isxdigit

Syntax

```
#include <ctype.h>
int isxdigit( int ch );
```

The function isxdigit() returns non-zero if its argument is a hexidecimal digit (i.e. A-F, a-f, or 0-9). Otherwise, zero is returned.

Standard C string and character function: memchr

Syntax

```
#include <string.h>
void *memchr( const void *buffer, int ch, size_t count );
```

The memchr() function looks for the first occurrence of *ch* within *count* characters in the array pointed to by *buffer*. The return value points to the location of the first occurrence of *ch*, or **NULL** if *ch* isn't found. For example:

```
char names[] = "Alan Bob Chris X Dave";
if( memchr(names,'X',strlen(names)) == NULL )
  printf( "Didn't find an X\n" );
else
  printf( "Found an X\n" );
```

Standard C string and character function: memcmp

Syntax

```
#include <string.h>
int memcmp( const void *buffer1, const void *buffer2, size_t count );
```

The function memcmp() compares the first *count* characters of *buffer1* and *buffer2*. The return values are as follows:

Value	Explanation
less than 0	buffer1 is less than buffer2
equal to 0	buffer1 is equal to buffer2
greater than 0	buffer1 is greater than buffer2

Standard C string and character function: memcpy

Syntax

```
#include <string.h>
void *memcpy( void *to, const void *from, size_t count );
```

The function memcpy() copies *count* characters from the array *from* to the array *to*. The return value of memcpy() is *to*. The behavior of memcpy() is undefined if *to* and *from* overlap.

Standard C string and character function: memmove

Syntax

```
#include <string.h>
void *memmove( void *to, const void *from, size_t count );
```

The memmove() function is identical to memcpy(), except that it works even if *to* and *from* overlap.

Standard C string and character function: memset

Syntax

```
#include <string.h>
void* memset( void* buffer, int ch, size_t count );
```

The function memset() copies *ch* into the first *count* characters of *buffer*, and returns *buffer*. memset() is useful for intializing a section of memory to some value. For example, this command:

```
const int ARRAY_LENGTH;
char the_array[ARRAY_LENGTH];
...
// zero out the contents of the_array
memset( the_array, '\0', ARRAY_LENGTH );
```

...is a very efficient way to set all values of the_array to zero.

The table below compares two different methods for initializing an array of characters: a for-loop versus memset(). As the size of the data being initialized increases, memset() clearly gets the job done much more quickly:

Input size	Initialized with a for-loop	Initialized with memset()
1000	0.016	0.017
10000	0.055	0.013
100000	0.443	0.029
1000000	4.337	0.291

Standard C string and character function: strcat

Syntax

```
#include <string.h>
char *strcat( char *str1, const char *str2 );
```

The strcat() function concatenates *str2* onto the end of *str1*, and returns *str1*. For example:

```
printf( "Enter your name: " );
scanf( "%s", name );
title = strcat( name, " the Great" );
printf( "Hello, %s\n", title );
```

Note that strcat() does not perform bounds checking, and thus risks overrunning *str1* or *str2*. For a similar (and safer) function that includes bounds checking, see strncat().

Another set of related (but non-standard) functions are strlcpy and strlcat.

Standard C string and character function: strchr

Syntax

```
#include <string.h>
char *strchr( const char *str, int ch );
```

The function strchr() returns a pointer to the first occurence of *ch* in *str*, or **NULL** if *ch* is not found.

Standard C string and character function: strcmp

Syntax

```
#include <string.h>
int strcmp( const char *str1, const char *str2 );
```

The function strcmp() compares *str1* and *str2*, then returns:

Return value	Explanation
less than 0	"str1" is less than "str2"
equal to 0	"str1" is equal to "str2"
greater than 0	"str1" is greater than "str2"

For example:

```
printf( "Enter your name: " );
scanf( "%s", name );
if( strcmp( name, "Mary" ) == 0 )
{
  printf( "Hello, Dr. Mary!\n" );
}
```

Note that if *str1* or *str2* are missing a null-termination character, then strcmp() may not produce valid results. For a similar (and safer) function that includes explicit bounds checking, see strncmp().

Standard C string and character function: strcoll

Syntax

```
#include <string.h>
int strcoll( const char *str1, const char *str2 );
```

The strcoll() function compares *str1* and *str2*, much like strcmp(). However, strcoll() performs the comparison using the locale specified by the (Standard C Date & Time) setlocale() function.

(Standard C Date &

Standard C string and character function: strcpy

Syntax

```
#include <string.h>
char *strcpy( char *to, const char *from );
```

The strcpy() function copies characters in the string *from* to the string *to*, including the null termination. The return value is *to*. Note that strcpy() does not perform bounds checking, and thus risks overrunning *from* or *to*. For a similar (and safer) function that includes bounds checking, see strncpy().

Another set of related (but non-standard) functions are strlcpy and strlcat.

Standard C string and character function: strcspn

Syntax

```
#include <string.h>
size_t strcspn( const char *str1, const char *str2 );
```

The function strcspn() returns the index of the first character in *str1* that matches any of the characters in *str2*.

Standard C string and character function: strerror

Syntax

```
#include <string.h>
char *strerror( int num );
```

The function strerror() returns an implementation defined string corresponding to *num*.

Standard C string and character function: strlen

Syntax

```
#include <string.h>
size_t strlen( char *str );
```

The strlen() function returns the length of `str` (determined by the number of characters before null termination).

Standard C string and character function: strncat

Syntax

```
#include <string.h>
char *strncat( char *str1, const char *str2, size_t count );
```

The function strncat() concatenates at most `count` characters of `str2` onto `str1`, adding a null termination. The resulting string is returned.

Another set of related (but non-standard) functions are strlcpy and strlcat.

Standard C string and character function: strncmp

Syntax

```
#include <string.h>
int strncmp( const char *str1, const char *str2, size_t count );
```

The strncmp() function compares at most `count` characters of `str1` and `str2`. The return value is as follows:

Return value	Explanation
less than 0	"str1" is less than "str2"
equal to 0	"str1" is equal to "str2"
greater than 0	"str1" is greater than str2"

If there are less than `count` characters in either string, then the comparison will stop after the first null termination is encountered.

Standard C string and character function: strncpy

Syntax

```
#include <string.h>
char *strncpy( char *to, const char *from, size_t count );
```

The strncpy() function copies at most `count` characters of `from` to the string `to`. If from has less than `count` characters, the remainder is padded with '\0' characters. The return value is the resulting string.

Another set of related (but non-standard) functions are strlcpy and strlcat.

Standard C string and character function: strpbrk

Syntax

```
#include <string.h>
char* strpbrk( const char* str1, const char* str2 );
```

The function strpbrk() returns a pointer to the first ocurrence in *str1* of any character in *str2*, or **NULL** if no such characters are present.

Standard C string and character function: strrchr

Syntax

```
#include <string.h>
char *strrchr( const char *str, int ch );
```

The function strrchr() returns a pointer to the last occurrence of *ch* in *str*, or **NULL** if no match is found.

Standard C string and character function: strspn

Syntax

```
#include <string.h>
size_t strspn( const char *str1, const char *str2 );
```

The strspn() function returns the index of the first character in *str1* that doesn't match any character in *str2*.

Standard C string and character function: strstr

Syntax

```
#include <string.h>
char *strstr( const char *str1, const char *str2 );
```

The function strstr() returns a pointer to the first occurrence of *str2* in *str1*, or **NULL** if no match is found. If the length of *str2* is zero, then strstr() will simply return *str1*.

For example, the following code checks for the existence of one string within another string:

```
char* str1 = "this is a string of characters";
char* str2 = "a string";
char* result = strstr( str1, str2 );
if( result == NULL ) printf( "Could not find '%s' in '%s'\n", str2, str1 );
else printf( "Found a substring: '%s'\n", result );
```

When run, the above code displays this output:

```
Found a substring: 'a string of characters'
```

Standard C string and character function: strtod

Syntax

```
#include <stdlib.h>
double strtod( const char *start, char **end );
```

The function strtod() returns whatever it encounters first in `start` as a double. *end* is set to point at whatever is left in `start` after that double. If overflow occurs, strtod() returns either **HUGE_VAL** or -**HUGE_VAL**.

Standard C string and character function: strtok

Syntax

```
#include <string.h>
char *strtok( char *str1, const char *str2 );
```

The strtok() function returns a pointer to the next "token" in `str1`, where `str2` contains the delimiters that determine the token. strtok() returns **NULL** if no token is found. In order to convert a string to tokens, the first call to strtok() should have `str1` point to the string to be tokenized. All calls after this should have `str1` be **NULL**.

For example:

```
char str[] = "now # is the time for all # good men to come to the # aid of their country";
char delims[] = "#";
char *result = NULL;
result = strtok( str, delims );
while( result != NULL )
{
    printf( "result is \"%s\"\n", result );
    result = strtok( NULL, delims );
}
```

The above code will display the following output:

```
result is "now "
result is " is the time for all "
result is " good men to come to the "
result is " aid of their country"
```

Standard C string and character function: strtol

Syntax

```
#include <stdlib.h>
long strtol( const char *start, char **end, int base );
```

The strtol() function returns whatever it encounters first in `start` as a long, doing the conversion to `base` if necessary. *end* is set to point to whatever is left in `start` after the long. If the result can not be represented by a long, then strtol() returns either **LONG_MAX** or **LONG_MIN**. Zero is returned upon error.

Standard C string and character function: strtoul

Syntax

```
#include <stdlib.h>
unsigned long strtoul( const char *start, char **end, int base );
```

The function strtoul() behaves exactly like strtol(), except that it returns an unsigned long rather than a mere long.

Standard C string and character function: strxfrm

Syntax

```
#include <string.h>
size_t strxfrm( char *str1, const char *str2, size_t num );
```

The strxfrm() function manipulates the first *num* characters of *str2* and stores them in *str1*. The result is such that if a strcoll() is performed on *str1* and the old *str2*, you will get the same result as with a strcmp().

Standard C string and character function: tolower

Syntax

```
#include <ctype.h>
int tolower( int ch );
```

The function tolower() returns the lowercase version of the character *ch*.

Standard C string and character function: toupper

Syntax

```
#include <ctype.h>
int toupper( int ch );
```

The toupper() function returns the uppercase version of the character *ch*.

Other Standard C Functions

The following is a list of all other standard C functions.

abort	stops the program
assert	stops the program if an expression isn't true
atexit	sets a function to be called when the program exits
bsearch	perform a binary search
exit	stop the program
getenv	get enviornment information about a variable
longjmp	start execution at a certain point in the program
qsort	perform a quicksort
raise	send a signal to the program
rand	returns a pseudorandom number
setjmp	set execution to start at a certain point
signal	register a function as a signal handler
srand	initialize the random number generator
system	perform a system call
va_arg	use variable length parameter lists

Standard C function: abort

Syntax

```
#include <stdlib.h>
void abort( void );
```

The function abort() terminates the current program. Depending on the implementation, the return value can indicate failure.

Standard C function: assert

Syntax

```
#include <assert.h>
assert( exp );
```

The assert() macro is used to test for errors. If *exp* evaluates to zero, assert() writes information to **stderr** and exits the program. If the macro NDEBUG is defined, the assert() macros will be ignored.

Standard C function: atexit

Syntax

```
#include <stdlib.h>
int atexit( void (*func)(void) );
```

The function atexit() causes the function pointed to by *func* to be called when the program terminates. You can make multiple calls to atexit() (at least 32, depending on your compiler) and those functions will be called in reverse order of their establishment. The return value of atexit() is zero upon success, and non-zero on failure.

Standard C function: bsearch

Syntax

```
  #include <stdlib.h>
  void *bsearch( const void *key, const void *buf, size_t num, size_t size, int
(*compare)(const void *, const void *) );
```

The bsearch() function searches *buf[0]* to *buf[num-1]* for an item that matches *key*, using a binary search. The function *compare* should return negative if its first argument is less than its second, zero if equal, and positive if greater. The items in the array *buf* should be in ascending order. The return value of bsearch() is a pointer to the matching item, or **NULL** if none is found.

Standard C function: exit

Syntax

```
#include <stdlib.h>
void exit( int exit_code );
```

The exit() function stops the program. *exit_code* is passed on to be the return value of the program, where usually zero indicates success and non-zero indicates an error.

Standard C function: getenv

Syntax

```
#include <stdlib.h>
char *getenv( const char *name );
```

The function getenv() returns environmental information associated with *name*, and is very implementation dependent. **NULL** is returned if no information about *name* is available.

Standard C function: longjmp

Syntax

```
#include <setjmp.h>
void longjmp( jmp_buf envbuf, int status );
```

The function longjmp() causes the program to start executing code at the point of the last call to setjmp(). *envbuf* is usually set through a call to setjmp(). *status* becomes the return value of setjmp() and can be used to figure out where longjmp() came from. *status* should not be set to zero.

Standard C function: qsort

Syntax

```
#include <stdlib.h>
void qsort( void *buf, size_t num, size_t size, int (*compare)(const void *, const void *)
);
```

The qsort() function sorts *buf* (which contains *num* items, each of size *size*) using Quicksort. The *compare* function is used to compare the items in *buf*. *compare* should return negative if the first argument is less than the second, zero if they are equal, and positive if the first argument is greater than the second. qsort() sorts *buf* in ascending order.

Example

For example, the following bit of code uses qsort() to sort an array of integers:

```
int compare_ints( const void* a, const void* b )
{
  int* arg1 = (int*) a;
  int* arg2 = (int*) b;
  if( *arg1 < *arg2 ) return -1;
  else if( *arg1 == *arg2 ) return 0;
  else return 1;
}

int array[] = { -2, 99, 0, -743, 2, 3, 4 };
int array_size = 7;

...

printf( "Before sorting: " );
for( int i = 0; i < array_size; i++ )
{
  printf( "%d ", array[i] );
}
printf( "\n" );

qsort( array, array_size, sizeof(int), compare_ints );

printf( "After sorting: " );
for( int i = 0; i < array_size; i++ )
{
  printf( "%d ", array[i] );
}
printf( "\n" );
```

When run, this code displays the following output:

```
Before sorting: -2 99 0 -743 2 3 4
After sorting: -743 -2 0 2 3 4 99
```

Standard C function: raise

Syntax

```
#include <signal.h>
int raise( int signal );
```

The raise() function sends the specified *signal* to the program. Some signals:

Signal	Meaning
SIGABRT	Termination error
SIGFPE	Floating pointer error
SIGILL	Bad instruction
SIGINT	User presed CTRL-C
SIGSEGV	Illegal memory access
SIGTERM	Terminate program

The return value is zero upon success, nonzero on failure.

Standard C function: rand

Syntax

```
#include <stdlib.h>
int rand( void );
```

The function rand() returns a pseudorandom integer between zero and RAND_MAX. An example:

```
srand( time(NULL) );
for( i = 0; i < 10; i++ )
  printf( "Random number #%d: %d\n", i, rand() );
```

Standard C function: setjmp

Syntax

```
#include <setjmp.h>
int setjmp( jmp_buf envbuf );
```

The setjmp() function saves the system stack in *envbuf* for use by a later call to longjmp(). When you first call setjmp(), its return value is zero. Later, when you call longjmp(), the second argument of longjmp() is what the return value of setjmp() will be. Confused? Read about longjmp().

Standard C function: signal

Syntax

```
#include <signal.h>
void ( *signal( int signal, void (* func) (int)) ) (int);
```

The signal() function sets `func` to be called when `signal` is recieved by your program. `func` can be a custom signal handler, or one of these macros (defined in signal.h):

Macro	Explanation
SIG_DFL	default signal handling
SIG_IGN	ignore the signal

Some basic signals that you can attach a signal handler to are:

Signal	Description
SIGTERM	Generic stop signal that can be caught.
SIGINT	Interrupt program, normally ctrl-c.
SIGQUIT	Interrupt program, similar to SIGINT.
SIGKILL	Stops the program. Cannot be caught.
SIGHUP	Reports a disconnected terminal.

The return value of signal() is the address of the previously defined function for this signal, or SIG_ERR is there is an error.

Example

The following example uses the signal() function to call an arbitrary number of functions when the user aborts the program. The functions are stored in a vector, and a single "clean-up" function calls each function in that vector of functions when the program is aborted:

```
void f1()
{
  printf("calling f1()...\n");
}

void f2()
{
  printf("calling f2()...\n");
}

typedef void(*endFunc)(void);
vector<endFunc> endFuncs;

void cleanUp( int dummy )
{
  for( unsigned int i = 0; i < endFuncs.size(); i++ )
  {
    endFunc f = endFuncs.at(i);
    (*f)();
  }
  exit(-1);
}
```

```
int main()
{

  // connect various signals to our clean-up function
  signal( SIGTERM, cleanUp );
  signal( SIGINT, cleanUp );
  signal( SIGQUIT, cleanUp );
  signal( SIGHUP, cleanUp );

  // add two specific clean-up functions to a list of functions
  endFuncs.push_back( f1 );
  endFuncs.push_back( f2 );

  // loop until the user breaks
  while( 1 );

  return 0;
}
```

Standard C function: srand

Syntax

```
#include <stdlib.h>
void srand( unsigned seed );
```

The function srand() is used to seed the random sequence generated by rand(). For any given *seed*, rand() will generate a specific "random" sequence over and over again.

```
srand( time(NULL) );
for( i = 0; i < 10; i++ )
  printf( "Random number #%d: %d\n", i, rand() );
```

(Standard C Date &

Standard C function: system

Syntax

```
#include <stdlib.h>
int system( const char *command );
```

The system() function runs the given *command* by passing it to the default command interpreter.

The return value is usually zero if the command executed without errors. If *command* is **NULL**, system() will test to see if there is a command interpreter available. Non-zero will be returned if there is a command interpreter available, zero if not.

Standard C function: va_arg, va_list, va_start, and va_end

Syntax

```
#include <stdarg.h>
type va_arg( va_list argptr, type );
void va_end( va_list argptr );
void va_start( va_list argptr, last_parm );
```

The va_arg() macros are used to pass a variable number of arguments to a function.

1. First, you must have a call to `va_start()` passing a valid **va_list** and the mandatory first argument of the function. This first argument can be anything; one way to use it is to have it be an integer describing the number of parameters being passed.
2. Next, you call `va_arg()` passing the **va_list** and the type of the argument to be returned. The return value of `va_arg()` is the current parameter.
3. Repeat calls to `va_arg()` for however many arguments you have.
4. Finally, a call to `va_end()` passing the **va_list** is necessary for proper cleanup.

For example:

```
int sum( int num, ... )
{
  int answer = 0;
  va_list argptr;

  va_start( argptr, num );

  for( ; num > 0; num-- )
  {
    answer += va_arg( argptr, int );
  }

  va_end( argptr );

  return( answer );
}

int main( void )
{
  int answer = sum( 4, 4, 3, 2, 1 );
  printf( "The answer is %d\n", answer );

  return( 0 );
}
```

This code displays 10, which is 4+3+2+1.

Here is another example of variable argument function, which is a simple printing function:

```
void my_printf( char *format, ... )
{
  va_list argptr;

  va_start( argptr, format );

  while( *format != '\0' )
  {
    // string
    if( *format == 's' )
    {
      char* s = va_arg( argptr, char * );
      printf( "Printing a string: %s\n", s );
    }
    // character
    else if( *format == 'c' )
    {
      char c = (char) va_arg( argptr, int );
      printf( "Printing a character: %c\n", c );
      break;
    }
    // integer
    else if( *format == 'd' )
    {
      int d = va_arg( argptr, int );
      printf( "Printing an integer: %d\n", d );
    }

    *format++;
  }
  va_end( argptr );
}


int main( void )
{
  my_printf( "sdc", "This is a string", 29, 'X' );

  return( 0 );
}
```

This code displays the following output when run:

```
Printing a string: This is a string
Printing an integer: 29
Printing a character: X
```

C++

This section contains a description of the C++ specific containers, iterators, exceptions and library functions.

C++ Containers

The C++ Containers (vectors, lists, etc.) are generic vessels capable of holding many different types of data. For example, the following statement creates a vector of integers:

```
vector<int> v;
```

Containers can hold standard objects (like the **int** in the above example) as well as custom objects, as long as the objects in the container meet a few requirements:

- The object must have a default constructor,
- an accessible destructor, and
- an accessible assignment operator.

When describing the functions associated with these various containers, this website defines the word **TYPE** to be the object type that the container holds. For example, in the above statement, **TYPE** would be **int**. Similarily, when referring to containers associated with pairs of data **key_type** and **value_type** are used to refer to the key and value types for that container.

C++ Iterators

Iterators are used to access members of the container classes, and can be used in a similar manner to pointers. For example, one might use an iterator to step through the elements of a vector. There are several different types of iterators:

Iterator	Description
input_iterator	Read values with forward movement. These can be incremented, compared, and dereferenced.
output_iterator	Write values with forward movement. These can be incremented and dereferenced.
forward_iterator	Read or write values with forward movement. These combine the functionality of input and output iterators with the ability to store the iterators value.
bidirectional_iterator	Read and write values with forward and backward movement. These are like the forward iterators, but you can increment and decrement them.
random_iterator	Read and write values with random access. These are the most powerful iterators, combining the functionality of bidirectional iterators with the ability to do pointer arithmetic and pointer comparisons.
reverse_iterator	Either a random iterator or a bidirectional iterator that moves in reverse direction.

Each of the container classes is associated with a type of iterator, and each of the STL algorithms uses a certain type of iterator. For example, vectors are associated with **random-access iterators**, which means that they can use algorithms that require random access. Since random-access iterators encompass all of the characteristics of the other iterators, vectors can use algorithms designed for other iterators as well.

The following code creates and uses an iterator with a vector:

```
vector<int> the_vector;
vector<int>::iterator the_iterator;
for( int i=0; i < 10; i++ )
  the_vector.push_back(i);
int total = 0;
the_iterator = the_vector.begin();
```

```
while( the_iterator != the_vector.end() )
{
  total += *the_iterator;
  the_iterator++;
}
cout << "Total=" << total << endl;
```

Notice that you can access the elements of the container by dereferencing the iterator.

C++ Exceptions

The <exception> header provides functions and classes for exception control. One basic class is `exception`:

```
class exception

{
  public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

The <stdexcept> header provides a small hierarchy of exception classes that can be thrown or caught:

- `exception`
  - `logic_error`
    - `domain_error`
    - `invalid_argument`
    - `length_error`
    - `out_of_range`
  - `runtime_error`
    - `range_error`
    - `overflow_error`
    - `underflow_error`

Logic errors are thrown if the program has internal errors that are caused by the user of a function. And in theory preventable.

Run-time errors are thrown if the cause is beyond the program and can't be predicted by the user of a function.

C++ I/O Functions

The <iostream> library automatically defines a few standard objects:

- cout, an object of the ostream class, which displays data to the standard output device.
- cerr, another object of the ostream class that writes unbuffered output to the standard error device.
- clog, like cerr, but uses buffered output.
- cin, an object of the istream class that reads data from the standard input device.

The <fstream> library allows programmers to do file input and output with the ifstream and ofstream classes.

C++ programmers can also do input and output from strings by using the String Stream class.

Some of the behavior of the C++ I/O streams (precision, justification, etc) may be modified by manipulating various io stream format flags.

The next section contains some examples of what you can do with C++ I/O.

bad	true if an error occurred
clear	clear and set status flags
close	close a stream
eof	true if at the end-of-file
fail	true if an error occurred
fill	manipulate the default fill character
flags	access or manipulate io stream format flags
flush	empty the buffer
gcount	number of characters read during last input
get	read characters
getline	read a line of characters
good	true if no errors have occurred
ignore	read and discard characters
open	open a new stream
peek	check the next input character
precision	manipulate the precision of a stream
put	write characters
putback	return characters to a stream
rdstate	returns the state flags of the stream
read	read data into a buffer
seekg	perform random access on an input stream
seekp	perform random access on output streams
setf	set format flags
sync_with_stdio	synchronize with standard I/O
tellg	read input stream pointers
tellp	read output stream pointers
unsetf	clear io stream format flags

width	access and manipulate the minimum field width
write	write characters

C++ I/O Examples

Reading From Files

Assume that we have a file named *`data.txt`* that contains this text:

```
Fry: One Jillion dollars.
[Everyone gasps.]
Auctioneer: Sir, that's not a number.
[Everyone gasps.]
```

We could use this code to read data from the file, word by word:

```
ifstream fin("data.txt");
string s;
while( fin >> s )
{
  cout << "Read from file: " << s << endl;
}
```

When used in this manner, we'll get space-delimited bits of text from the file:

```
Read from file: Fry:
Read from file: One
Read from file: Jillion
Read from file: dollars.
Read from file: [Everyone
Read from file: gasps.]
Read from file: Auctioneer:
Read from file: Sir,
Read from file: that's
Read from file: not
Read from file: a
Read from file: number.
Read from file: [Everyone
Read from file: gasps.]
```

Note that in the previous example, all of the whitespace that separated words (including newlines) was lost. If we were interested in preserving whitespace, we could read the file in line-by-line using the I/O getline() function.

```
ifstream fin("data.txt");
const int LINE_LENGTH = 100;
char str[LINE_LENGTH];

while( fin.getline(str,LINE_LENGTH) )
{
  cout << "Read from file: " << str << endl;
}
```

Reading line-by-line produces the following output:

```
Read from file: Fry: One Jillion dollars.
Read from file: [Everyone gasps.]
Read from file: Auctioneer: Sir, that's not a number.
Read from file: [Everyone gasps.]
```

If you want to avoid reading into character arrays, you can use the C++ string getline() function to read lines into strings:

```
ifstream fin("data.txt");
string s;
while( getline(fin,s) )
{
  cout << "Read from file: " << s << endl;
}
```

Checking For Errors

Simply evaluating an I/O object in a boolean context will return false if any errors have occurred:

```
string filename = "data.txt";
ifstream fin( filename.c_str() );
if( !fin )
{
  cout << "Error opening " << filename << " for input" << endl;
  exit(-1);
}
```

I/O Constructors

Syntax

```
#include <fstream>
fstream( const char *filename, openmode mode );
ifstream( const char *filename, openmode mode );
ofstream( const char *filename, openmode mode );
```

The fstream, ifstream, and ofstream objects are used to do file I/O. The optional *mode* defines how the file is to be opened, according to the io stream mode flags. The optional *filename* specifies the file to be opened and associated with the stream.

Input and output file streams can be used in a similar manner to C++ predefined I/O streams, cin and cout.

Example

The following code reads input data and appends the result to an output file.

```
ifstream fin( "/tmp/data.txt" );
ofstream fout( "/tmp/results.txt", ios::app );
while( fin >> temp )
  fout << temp + 2 << endl;
fin.close();
fout.close();
```

C++ I/O Flags

Format flags

C++ defines some format flags for standard input and output, which can be manipulated with the flags(), setf(), and unsetf() functions. For example,

```
cout.setf(ios::left);
```

turns on left justification for all output directed to **cout**.

Flag	Meaning
boolalpha	Boolean values can be input/output using the words "true" and "false".
dec	Numeric values are displayed in decimal.
fixed	Display floating point values using normal notation (as opposed to scientific).
hex	Numeric values are displayed in hexidecimal.
internal	If a numeric value is padded to fill a field, spaces are inserted between the sign and base character.
left	Output is left justified.
oct	Numeric values are displayed in octal.
right	Output is right justified.
scientific	Display floating point values using scientific notation.
showbase	Display the base of all numeric values.
showpoint	Display a decimal and extra zeros, even when not needed.
showpos	Display a leading plus sign before positive numeric values.
skipws	Discard whitespace characters (spaces, tabs, newlines) when reading from a stream.
unitbuf	Flush the buffer after each insertion.
uppercase	Display the "e" of scientific notation and the "x" of hexidecimal notation as capital letters.

Manipulators

You can also manipulate flags indirectly, using the following *manipulators*. Most programmers are familiar with the **endl** manipulator, which might give you an idea of how manipulators are used. For example, to set the *dec* flag, you might use the following command:

```
cout << dec;
```

Manipulators defined in <iostream>

Manipulator	Description	Input	Output
boolalpha	Turns on the boolalpha flag	X	X
dec	Turns on the dec flag	X	X
endl	Output a newline character, flush the stream		X
ends	Output a null character		X
fixed	Turns on the fixed flag		X
flush	Flushes the stream		X
hex	Turns on the hex flag	X	X
internal	Turns on the internal flag		X
left	Turns on the left flag		X
noboolalpha	Turns off the boolalpha flag	X	X
noshowbase	Turns off the showbase flag		X
noshowpoint	Turns off the showpoint flag		X
noshowpos	Turns off the showpos flag		X
noskipws	Turns off the skipws flag	X	
nounitbuf	Turns off the unitbuf flag		X
nouppercase	Turns off the uppercase flag		X
oct	Turns on the oct flag	X	X
right	Turns on the right flag		X
scientific	Turns on the scientific flag		X
showbase	Turns on the showbase flag		X
showpoint	Turns on the showpoint flag		X
showpos	Turns on the showpos flag		X
skipws	Turns on the skipws flag	X	
unitbuf	Turns on the unitbuf flag		X
uppercase	Turns on the uppercase flag		X
ws	Skip any leading whitespace	X	

Manipulators defined in <iomanip>

Manipulator	Description	Input	Output
resetiosflags(long f)	Turn off the flags specified by f	X	X
setbase(int base)	Sets the number base to base		X
setfill(int ch)	Sets the fill character to ch		X
setiosflags(long f)	Turn on the flags specified by f	X	X
setprecision(int p)	Sets the number of digits of precision		X
setw(int w)	Sets the field width to w		X

State flags

The I/O stream state flags tell you the current state of an I/O stream. The flags are:

Flag	Meaning
badbit	a fatal error has occurred
eofbit	EOF has been found
failbit	a nonfatal error has occurred
goodbit	no errors have occurred

Mode flags

The I/O stream mode flags allow you to access files in different ways. The flags are:

Mode	Meaning
ios::app	append output
ios::ate	seek to EOF when opened
ios::binary	open the file in binary mode
ios::in	open the file for reading
ios::out	open the file for writing
ios::trunc	overwrite the existing file

C++ I/O function: bad

Syntax

```
#include <fstream>
bool bad();
```

The bad() function returns true if a fatal error with the current stream has occurred, false otherwise.

C++ I/O function: clear

Syntax

```
#include <fstream>
void clear( iostate flags = ios::goodbit );
```

The function clear() does two things:

- it clears all io stream state flags associated with the current stream,
- and sets the flags denoted by *flags*

The *flags* argument defaults to ios::goodbit, which means that by default, all flags will be cleared and ios::goodbit will be set.

Example

For example, the following code uses the clear() function to reset the flags of an output file stream, after an attempt is made to read from that output stream:

```
fstream outputFile( "output.txt", fstream::out );

// try to read from the output stream; this shouldn't work
int val;
outputFile >> val;
if( outputFile.fail() )
{
  cout << "Error reading from the output stream" << endl;
  // reset the flags associated with the stream
  outputFile.clear();
}

for( int i = 0; i < 10; i++ )
{
  outputFile << i << " ";
}
outputFile << endl;
```

C++ I/O function: close

Syntax

```
#include <fstream>
void close();
```

The close() function closes the associated file stream.

C++ I/O function: eof

Syntax

```
#include <fstream>
bool eof();
```

The function eof() returns true if the end of the associated input file has been reached, false otherwise.

For example, the following code reads data from an input stream `in` and writes it to an output stream `out`, using eof() at the end to check if an error occurred:

```
char buf[BUFSIZE];
do
{
  in.read( buf, BUFSIZE );
  std::streamsize n = in.gcount();
  out.write( buf, n );
} while( in.good() );
if( in.bad() || !in.eof() )
{
  // fatal error occurred
}
in.close();
```

C++ I/O function: fail

Syntax

```
#include <fstream>
bool fail();
```

The fail() function returns true if an error has occurred with the current stream, false otherwise.

C++ I/O function: fill

Syntax

```
#include <fstream>
char fill();
char fill( char ch );
```

The function fill() either returns the current fill character, or sets the current fill character to `ch`.

The fill character is defined as the character that is used for padding when a number is smaller than the specified width(). The default fill character is the space character.

C++ I/O function: flags

Syntax

```
#include <fstream>
fmtflags flags();
fmtflags flags( fmtflags f );
```

The flags() function either returns the io stream format flags for the current stream, or sets the flags for the current stream to be `f`.

C++ I/O function: flush

Syntax

```
#include <fstream>
ostream& flush();
```

The flush() function causes the buffer for the current output stream to be actually written out to the attached device.

This function is useful for printing out debugging information, because sometimes programs abort before they have a chance to write their output buffers to the screen. Judicious use of flush() can ensure that all of your debugging statements actually get printed.

C++ I/O function: gcount

Syntax

```
#include <fstream>
streamsize gcount();
```

The function gcount() is used with input streams, and returns the number of characters read by the last input operation.

C++ I/O function: get

Syntax

```
#include <fstream>
int get();
istream& get( char& ch );
istream& get( char* buffer, streamsize num );
istream& get( char* buffer, streamsize num, char delim );
istream& get( streambuf& buffer );
istream& get( streambuf& buffer, char delim );
```

The get() function is used with input streams, and either:

- reads a character and returns that value,
- reads a character and stores it as *ch*,
- reads characters into *buffer* until *num* - 1 characters have been read, or **EOF** or newline encountered,
- reads characters into *buffer* until *num* - 1 characters have been read, or **EOF** or the *delim* character encountered (*delim* is not read until next time),
- reads characters into buffer until a newline or **EOF** is encountered,
- or reads characters into buffer until a newline, **EOF**, or *delim* character is encountered (again, *delim* isn't read until the next get()).

For example, the following code displays the contents of a file called temp.txt, character by character:

```
char ch;
ifstream fin( "temp.txt" );
while( fin.get(ch) )
  cout << ch;
fin.close();
```

C++ I/O function: getline

Syntax

```
#include <fstream>
istream& getline( char* buffer, streamsize num );
istream& getline( char* buffer, streamsize num, char delim );
```

The getline() function is used with input streams, and reads characters into *buffer* until either:

- *num* - 1 characters have been read,
- a newline is encountered,
- an **EOF** is encountered,
- or, optionally, until the character *delim* is read. The *delim* character is not put into buffer.

For example, the following code uses the getline function to display the first 100 characters from each line of a text file:

```
ifstream fin("tmp.dat");

int MAX_LENGTH = 100;
char line[MAX_LENGTH];

while( fin.getline(line, MAX_LENGTH) )
{
  cout << "read line: " << line << endl;
}
```

If you'd like to read lines from a file into strings instead of character arrays, consider using the string getline function.

C++ I/O function: good

Syntax

```
#include <fstream>
bool good();
```

The function good() returns true if no errors have occurred with the current stream, false otherwise.

C++ I/O function: ignore

Syntax

```
#include <fstream>
istream& ignore( streamsize num=1, int delim=EOF );
```

The ignore() function is used with input streams. It reads and throws away characters until *num* characters have been read (where *num* defaults to 1) or until the character *delim* is read (where *delim* defaults to **EOF**).

The ignore() function can sometimes be useful when using the getline() function together with the >> operator. For example, if you read some input that is followed by a newline using the >> operator, the newline will remain in the input as the next thing to be read. Since getline() will by default stop reading input when it reaches a newline, a subsequent call to getline() will return an empty string. In this case, the ignore() function could be called before getline() to "throw away" the newline.

C++ I/O function: open

Syntax

```
#include <fstream>
void open( const char *filename );
void open( const char *filename, openmode mode = default_mode );
```

The function open() is used with file streams. It opens `filename` and associates it with the current stream. The optional io stream mode flag `mode` defaults to ios::in for ifstream, ios::out for ofstream, and ios::in|ios::out for fstream.

If open() fails, the resulting stream will evaluate to false when used in a Boolean expression. For example:

```
ifstream inputStream;
inputStream.open("file.txt");
if( !inputStream )
{
  cerr << "Error opening input stream" << endl;
  return;
}
```

C++ I/O function: peek

Syntax

```
#include <fstream>
int peek();
```

The function peek() is used with input streams, and returns the next character in the stream or **EOF** if the end of file is read. peek() does not remove the character from the stream.

C++ I/O function: precision

Syntax

```
#include <fstream>
streamsize precision();
streamsize precision( streamsize p );
```

The precision() function either sets or returns the current number of digits that is displayed for floating-point variables.

For example, the following code sets the precision of the cout stream to 5:

```
float num = 314.15926535;
cout.precision( 5 );
cout << num;
```

This code displays the following output:

```
314.16
```

C++ I/O function: put

Syntax

```
#include <fstream>
ostream& put( char ch );
```

The function put() is used with output streams, and writes the character *ch* to the stream.

C++ I/O function: putback

Syntax

```
#include <fstream>
istream& putback( char ch );
```

The putback() function is used with input streams, and returns the previously-read character *ch* to the input stream.

C++ I/O function: rdstate

Syntax

```
#include <fstream>
iostate rdstate();
```

The rdstate() function returns the io stream state flags of the current stream.

C++ I/O function: read

Syntax

```
#include <fstream>
istream& read( char* buffer, streamsize num );
```

The function read() is used with input streams, and reads *num* bytes from the stream before placing them in *buffer*. If **EOF** is encountered, read() stops, leaving however many bytes it put into *buffer* as they are.

For example:

```
struct
{
  int height;
  int width;
} rectangle;

input_file.read( (char *)(&rectangle), sizeof(rectangle) );
if( input_file.bad() )
{
  cerr << "Error reading data" << endl;
  exit( 0 );
}
```

C++ I/O function: seekg

Syntax

```
#include <fstream>
istream& seekg( off_type offset, ios::seekdir origin );
istream& seekg( pos_type position );
```

The function seekg() is used with input streams, and it repositions the "get" pointer for the current stream to *`offset`* bytes away from *`origin`*, or places the "get" pointer at *`position`*.

C++ I/O function: seekp

Syntax

```
#include <fstream>
ostream& seekp( off_type offset, ios::seekdir origin );
ostream& seekp( pos_type position );
```

The seekp() function is used with output streams, but is otherwise very similar to seekg().

C++ I/O function: setf

Syntax

```
#include <fstream>
fmtflags setf( fmtflags flags );
fmtflags setf( fmtflags flags, fmtflags needed );
```

The function setf() sets the io stream format flags of the current stream to *`flags`*. The optional *needed* argument specifies that only the flags that are in both *`flags`* and *needed* should be set. The return value is the previous configuration of io stream format flags.

For example:

```
int number = 0x3FF;
cout.setf( ios::dec );
cout << "Decimal: " << number << endl;
cout.unsetf( ios::dec );
cout.setf( ios::hex );
cout << "Hexadecimal: " << number << endl;
```

Note that the preceding code is functionally identical to:

```
int number = 0x3FF;
cout << "Decimal: " << number << endl << hex << "Hexadecimal: " << number << dec << endl;
```

thanks to io stream manipulators.

C++ I/O function: sync_with_stdio

Syntax

```
#include <fstream>
static bool sync_with_stdio( bool sync=true );
```

The sync_with_stdio() function allows you to turn on and off the ability for the C++ I/O system to work with the C I/O system.

C++ I/O function: tellg

Syntax

```
#include <fstream>
pos_type tellg();
```

The tellg() function is used with input streams, and returns the current "get" position of the pointer in the stream.

C++ I/O function: tellp

Syntax

```
#include <fstream>
pos_type tellp();
```

The tellp() function is used with output streams, and returns the current "put" position of the pointer in the stream.

For example, the following code displays the file pointer as it writes to a stream:

```
string s("In Xanadu did Kubla Khan...");
ofstream fout("output.txt");
for( int i=0; i < s.length(); i++ )
{
  cout << "File pointer: " << fout.tellp();
  fout.put( s[i] );
  cout << " " << s[i] << endl;
}
fout.close();
```

C++ I/O function: unsetf

Syntax

```
#include <fstream>
void unsetf( fmtflags flags );
```

The function unsetf() uses *flags* to clear the io stream format flags associated with the current stream.

C++ I/O function: width

Syntax

```
#include <fstream>
int width();
int width( int w );
```

The function width() returns the current width, which is defined as the minimum number of characters to display with each output. The optional argument *w* can be used to set the width.

For example:

```
cout.width( 5 );
cout << "2";
```

displays

```
    2
```

(that's four spaces followed by a '2')

C++ I/O function: write

Syntax

```
#include <fstream>
ostream& write( const char* buffer, streamsize num );
```

The write() function is used with output streams, and writes *`num`* bytes from *`buffer`* to the current output stream.

C++ String Functions

append	append characters and strings onto a string
assign	give a string values from strings of characters and other C++ strings
at	returns the character at a specific location
begin	returns an iterator to the beginning of the string
c_str	returns a non-modifiable standard C character array version of the string
capacity	returns the number of characters that the string can hold
clear	removes all characters from the string
compare	compares two strings
copy	copies characters from a string into an array
data	returns a pointer to the first character of a string
empty	true if the string has no characters
end	returns an iterator just past the last character of a string
erase	removes characters from a string
find	find characters in the string
find_first_not_of	find first absence of characters
find_first_of	find first occurrence of characters
find_last_not_of	find last absence of characters
find_last_of	find last occurrence of characters
getline	read data from an I/O stream into a string
insert	insert characters into a string
length	returns the length of the string
max_size	returns the maximum number of characters that the string can hold
push_back	add a character to the end of the string
rbegin	returns a reverse_iterator to the end of the string
rend	returns a reverse_iterator to the beginning of the string
replace	replace characters in the string
reserve	sets the minimum capacity of the string
resize	change the size of the string
rfind	find the last occurrence of a substring
size	returns the number of items in the string
substr	returns a certain substring
swap	swap the contents of this string with another

String constructors

Syntax

```
#include <string>
string();
string( const string& s );
string( size_type length, const char& ch );
string( const char* str );
string( const char* str, size_type length );
string( const string& str, size_type index, size_type length );
string( input_iterator start, input_iterator end );
~string();
```

The string constructors create a new string containing:

- nothing; an empty string,
- a copy of the given string *s*,
- *length* copies of *ch*,
- a duplicate of *str* (optionally up to *length* characters long),
- a substring of *str* starting at *index* and *length* characters long
- a string of characterss denoted by the *start* and *end* iterators

For example,

```
string str1( 5, 'c' );
string str2( "Now is the time..." );
string str3( str2, 11, 4 );
cout << str1 << endl;
cout << str2 << endl;
cout << str3 << endl;
```

displays

```
ccccc
Now is the time...
time
```

The string constructors usually run in linear time, except the empty constructor, which runs in constant time.

String operators

Syntax

```
#include <string>
bool operator==(const string& c1, const string& c2);
bool operator!=(const string& c1, const string& c2);
bool operator<(const string& c1, const string& c2);
bool operator>(const string& c1, const string& c2);
bool operator<=(const string& c1, const string& c2);
bool operator>=(const string& c1, const string& c2);
string operator+(const string& s1, const string& s2 );
string operator+(const char* s, const string& s2 );
string operator+( char c, const string& s2 );
string operator+( const string& s1, const char* s );
string operator+( const string& s1, char c );
ostream& operator<<( ostream& os, const string& s );
istream& operator>>( istream& is, string& s );
string& operator=( const string& s );
string& operator=( const char* s );
string& operator=( char ch );
char& operator[]( size_type index );
```

C++ strings can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Performing a comparison or assigning one string to another takes linear time.

Two strings are equal if:

1. Their size is the same, and
2. Each member in location i in one string is equal to the the member in location i in the other `string.`

Comparisons among strings are done lexicographically.

In addition to the normal container operators, strings can also be concatenated with the + operator and fed to the C++ I/O stream classes with the << and >> operators.

For example, the following code concatenates two strings and displays the result:

```
string s1 = "Now is the time...";
string s2 = "for all good men...";
string s3 = s1 + s2;
cout << "s3 is " << s3 << endl;
```

Futhermore, strings can be assigned values that are other strings, character arrays, or even single characters. The following code is perfectly valid:

```
char ch = 'N';
string s;
s = ch;
```

Individual characters of a string can be examined with the [] operator, which runs in constant time.

C++ String function: append

Syntax

```
#include <string>
string& append( const string& str );
string& append( const char* str );
string& append( const string& str, size_type index, size_type len );
string& append( const char* str, size_type num );
string& append( size_type num, char ch );
string& append( input_iterator start, input_iterator end );
```

The append() function either:

- appends *str* on to the end of the current string,
- appends a substring of *str* starting at *index* that is *len* characters long on to the end of the current string,
- appends *num* characters of *str* on to the end of the current string,
- appends *num* repititions of *ch* on to the end of the current string,
- or appends the sequence denoted by *start* and *end* on to the end of the current string.

For example, the following code uses append() to add 10 copies of the '!' character to a string:

```
string str = "Hello World";
str.append( 10, '!' );
cout << str << endl;
```

That code displays:

```
Hello World!!!!!!!!!!
```

In the next example, append() is used to concatenate a substring of one string onto another string:

```
string str1 = "Eventually I stopped caring...";
string str2 = "but that was the '80s so nobody noticed.";

str1.append( str2, 25, 15 );
cout << "str1 is " << str1 << endl;
```

When run, the above code displays:

```
str1 is Eventually I stopped caring...nobody noticed.
```

C++ String function: assign

Syntax

```
#include <string>
void assign( size_type num, const char& val );
void assign( input_iterator start, input_iterator end );
string& assign( const string& str );
string& assign( const char* str );
string& assign( const char* str, size_type num );
string& assign( const string& str, size_type index, size_type len );
string& assign( size_type num, const char& ch );
```

The default assign() function gives the current string the values from *start* to *end*, or gives it *num* copies of *val*.

In addition to the normal assign functionality that all C++ containers have, strings possess an assign() function that also allows them to:

- assign *str* to the current string,
- assign the first *num* characters of *str* to the current string,
- assign a substring of *str* starting at *index* that is *len* characters long to the current string,

For example, the following code:

```
string str1, str2 = "War and Peace";
str1.assign( str2, 4, 3 );
cout << str1 << endl;
```

displays

```
and
```

This function will destroy the previous contents of the string.

C++ String function: at

Syntax

```
#include <string>
TYPE& at( size_type loc );
const TYPE& at( size_type loc ) const;
```

The at() function returns a reference to the element in the string at index *loc*. The at() function is safer than the [] operator, because it won't let you reference items outside the bounds of the string.

For example, consider the following code:

```
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ )
{
  cout << "Element " << i << " is " << v[i] << endl;
}
```

This code overrunns the end of the vector, producing potentially dangerous results. The following code would be much safer:

```
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ )
{
  cout << "Element " << i << " is " << v.at(i) << endl;
}
```

Instead of attempting to read garbage values from memory, the at() function will realize that it is about to overrun the vector and will throw an exception.

C++ String function: begin

Syntax

```
#include <string>
iterator begin();
const_iterator begin() const;
```

The function begin() returns an iterator to the first element of the string. begin() should run in constant time.

For example, the following code uses begin() to initialize an iterator that is used to traverse a list:

```
// Create a list of characters
list<char> charList;
for( int i=0; i < 10; i++ )
{
  charList.push_front( i + 65 );
}
// Display the list
list<char>::iterator theIterator;
for( theIterator = charList.begin(); theIterator != charList.end(); theIterator++ )
{
  cout << *theIterator;
}
```

C++ String function: c_str

Syntax

```
#include <string>
const char* c_str();
```

The function c_str() returns a const pointer to a regular C string, identical to the current string. The returned string is null-terminated.

Note that since the returned pointer is of type const, the character data that c_str() returns **cannot be modified**. Furthermore, you do not need to call free() or delete on this pointer.

C++ String function: capacity

Syntax

```
#include <string>
size_type capacity() const;
```

The capacity() function returns the number of elements that the string can hold before it will need to allocate more space.

For example, the following code uses two different methods to set the capacity of two vectors. One method passes an argument to the constructor that suggests an initial size, the other method calls the reserve function to achieve a similar goal:

```
vector<int> v1(10);
cout << "The capacity of v1 is " << v1.capacity() << endl;
vector<int> v2;
v2.reserve(20);
cout << "The capacity of v2 is " << v2.capacity() << endl;
```

When run, the above code produces the following output:

```
The capacity of v1 is 10
The capacity of v2 is 20
```

C++ containers are designed to grow in size dynamically. This frees the programmer from having to worry about storing an arbitrary number of elements in a container. However, sometimes the programmer can improve the performance of her program by giving hints to the compiler about the size of the containers that the program will use. These hints come in the form of the reserve() function and the constructor used in the above example, which tell the compiler how large the container is expected to get.

The capacity() function runs in constant time.

C++ String function: clear

Syntax

```
#include <string>
void clear();
```

The function clear() deletes all of the elements in the string. clear() runs in linear time.

C++ String function: compare

Syntax

```
#include <string>
int compare( const string& str );
int compare( const char* str );
int compare( size_type index, size_type length, const string& str );
int compare( size_type index, size_type length, const string& str, size_type index2,
size_type length2 );
int compare( size_type index, size_type length, const char* str, size_type length2 );
```

The compare() function either compares `str` to the current string in a variety of ways, returning

Return Value	Case
less than zero	this < str
zero	this == str
greater than zero	this > str

The various functions either:

- compare `str` to the current string,
- compare `str` to a substring of the current string, starting at *`index`* for *`length`* characters,
- compare a substring of `str` to a substring of the current string, where *`index2`* and *`length2`* refer to `str` and *`index`* and *`length`* refer to the current string,
- or compare a substring of `str` to a substring of the current string, where the substring of `str` begins at zero and is *`length2`* characters long, and the substring of the current string begins at *`index`* and is *`length`* characters long.

For example, the following code uses compare() to compare four strings with eachother:

```
string names[] = {"Homer", "Marge", "3-eyed fish", "inanimate carbon rod"};

for( int i = 0; i < 4; i++ )
{
  for( int j = 0; j < 4; j++ )
  {
    cout << names[i].compare( names[j] ) << " ";
  }
  cout << endl;
}
```

Data from the above code was used to generate this table, which shows how the various strings compare to eachother:

	Homer	Marge	3-eyed fish	inanimate carbon rod
"Homer".compare(x)	0	-1	1	-1
"Marge".compare(x)	1	0	1	-1
"3-eyed fish".compare(x)	-1	-1	0	-1
"inanimate carbon rod".compare(x)	1	1	1	0

C++ String function: copy

Syntax

```
#include <string>
size_type copy( char* str, size_type num, size_type index = 0 );
```

The copy() function copies *num* characters of the current string (starting at *index* if it's specified, 0 otherwise) into *str*.

The return value of copy() is the number of characters copied.

For example, the following code uses copy() to extract a substring of a string into an array of characters:

```
char buf[30];
memset( buf, '\0', 30 );
string str = "Trying is the first step towards failure.";
str.copy( buf, 24 );
cout << buf << endl;
```

When run, this code displays:

```
Trying is the first step
```

Note that before calling copy(), we first call (Standard C String and Character) memset() to fill the destination array with copies of the **NULL** character. This step is included to make sure that the resulting array of characters is **NULL**-terminated.

C++ String function: data

Syntax

```
#include <string>
const char *data();
```

The function data() returns a pointer to the first character in the current string.

C++ String function: empty

Syntax

```
#include <string>
bool empty() const;
```

The empty() function returns true if the string has no elements, false otherwise.

For example:

```
string s1;
string s2("");
string s3("This is a string");
cout.setf(ios::boolalpha);
cout << s1.empty() << endl;
cout << s2.empty() << endl;
cout << s3.empty() << endl;
```

When run, this code produces the following output:

```
true
true
false
```

C++ String function: end

Syntax

```
#include <string>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the string.

Note that before you can access the last element of the string using an iterator that you get from a call to end(), you'll have to decrement the iterator first.

For example, the following code uses begin() and end() to iterate through all of the members of a vector:

```
vector<int> v1( 5, 789 );
vector<int>::iterator it;
for( it = v1.begin(); it != v1.end(); it++ )
{
  cout << *it << endl;
}
```

The iterator is initialized with a call to begin(). After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling end(). Since end() returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

end() runs in constant time.

C++ String function: erase

Syntax

```
#include <string>
iterator erase( iterator loc );
iterator erase( iterator start, iterator end );
string& erase( size_type index = 0, size_type num = npos );
```

The erase() function either:

- removes the character pointed to by *loc*, returning an iterator to the next character,
- removes the characters between *start* and *end* (including the one at *start* but not the one at *end*), returning an iterator to the character after the last character removed,
- or removes *num* characters from the current string, starting at *index*, and returns *this.

The parameters *index* and *num* have default values, which means that erase() can be called with just *index* to erase all characters after *index* or with no arguments to erase all characters.

For example:

```
string s("So, you like donuts, eh? Well, have all the donuts in the world!");
cout << "The original string is '" << s << "'" << endl;

s.erase( 50, 14 );
cout << "Now the string is '" << s << "'" << endl;
s.erase( 24 );
cout << "Now the string is '" << s << "'" << endl;
s.erase();
cout << "Now the string is '" << s << "'" << endl;
```

will display

```
The original string is 'So, you like donuts, eh? Well, have all the donuts in the world!'
Now the string is 'So, you like donuts, eh? Well, have all the donuts'
Now the string is 'So, you like donuts, eh?'
Now the string is ''
```

erase() runs in linear time.

C++ String function: find

Syntax

```
#include <string>
size_type find( const string& str, size_type index );
size_type find( const char* str, size_type index );
size_type find( const char* str, size_type index, size_type length );
size_type find( char ch, size_type index );
```

The function find() either:

- returns the first occurrence of *str* within the current string, starting at *index*, string::npos if nothing is found,
- if the *length* parameter is given, then find() returns the first occurrence of the first *length* characters of *str* within the current string, starting at *index*, string::npos if nothing is found,
- or returns the index of the first occurrence *ch* within the current string, starting at *index*, string::npos if nothing is found.

For example:

```
string str1( "Alpha Beta Gamma Delta" );
string::size_type loc = str1.find( "Omega", 0 );
if( loc != string::npos )
{
  cout << "Found Omega at " << loc << endl;
}
else
{
  cout << "Didn't find Omega" << endl;
}
```

C++ String function: find_first_not_of

Syntax

```
#include <string>
size_type find_first_not_of( const string& str, size_type index = 0 );
size_type find_first_not_of( const char* str, size_type index = 0 );
size_type find_first_not_of( const char* str, size_type index, size_type num );
size_type find_first_not_of( char ch, size_type index = 0 );
```

The find_first_not_of() function either:

- returns the index of the first character within the current string that does not match any character in *str*, beginning the search at *index*, string::npos if nothing is found,
- searches the current string, beginning at *index*, for any character that does not match the first *num* characters in *str*, returning the index in the current string of the first character found that meets this criteria, otherwise returning string::npos,
- or returns the index of the first occurrence of a character that does not match *ch* in the current string, starting the search at *index*, string::npos if nothing is found.

For example, the following code searches a string of text for the first character that is not a lower-case character, space, comma, or hypen:

```
string lower_case = "abcdefghijklmnopqrstuvwxyz ,-";
string str = "this is the lower-case part, AND THIS IS THE UPPER-CASE PART";
cout << "first non-lower-case letter in str at: " << str.find_first_not_of(lower_case) << endl;
```

When run, find_first_not_of() finds the first upper-case letter in *str* at index 29 and displays this output:

```
first non-lower-case letter in str at: 29
```

C++ String function: find_first_of

Syntax

```
#include <string>
size_type find_first_of( const string &str, size_type index = 0 );
size_type find_first_of( const char* str, size_type index = 0 );
size_type find_first_of( const char* str, size_type index, size_type num );
size_type find_first_of( char ch, size_type index = 0 );
```

The find_first_of() function either:

- returns the index of the first character within the current string that matches any character in *str*, beginning the search at *index*, string::npos if nothing is found,
- searches the current string, beginning at *index*, for any of the first *num* characters in *str*, returning the index in the current string of the first character found, or string::npos if no characters match,
- or returns the index of the first occurrence of *ch* in the current string, starting the search at *index*, string::npos if nothing is found.

C++ String function: find_last_not_of

Syntax

```
#include <string>
size_type find_last_not_of( const string& str, size_type index = npos );
size_type find_last_not_of( const char* str, size_type index = npos);
size_type find_last_not_of( const char* str, size_type index, size_type num );
size_type find_last_not_of( char ch, size_type index = npos );
```

The find_last_not_of() function either:

- returns the index of the last character within the current string that does not match any character in *str*, doing a reverse search from *index*, string::npos if nothing is found,
- does a reverse search in the current string, beginning at *index*, for any character that does not match the first *num* characters in *str*, returning the index in the current string of the first character found that meets this criteria, otherwise returning string::npos,
- or returns the index of the last occurrence of a character that does not match *ch* in the current string, doing a reverse search from *index*, string::npos if nothing is found.

For example, the following code searches for the last non-lower-case character in a mixed string of characters:

```
string lower_case = "abcdefghijklmnopqrstuvwxyz";
string str = "abcdefgABCDEFGhijklmnop";
cout << "last non-lower-case letter in str at: " << str.find_last_not_of(lower_case) << endl;
```

This code displays the following output:

```
last non-lower-case letter in str at: 13
```

C++ String function: find_last_of

Syntax

```
#include <string>
size_type find_last_of( const string& str, size_type index = npos );
size_type find_last_of( const char* str, size_type index = npos );
size_type find_last_of( const char* str, size_type index, size_type num );
size_type find_last_of( char ch, size_type index = npos );
```

The find_last_of() function either:

- does a reverse search from *index*, returning the index of the first character within the current string that matches any character in *str*, or string::npos if nothing is found,
- does a reverse search in the current string, beginning at *index*, for any of the first *num* characters in *str*, returning the index in the current string of the first character found, or string::npos if no characters match,
- or does a reverse search from *index*, returning the index of the first occurrence of *ch* in the current string, string::npos if nothing is found.

C++ String function: getline

Syntax

```
#include <string>
istream& getline( istream& is, string& s, char delimiter = '\n' );
```

The C++ string class defines the global function getline() to read strings from an I/O stream. The getline() function, which is not part of the string class, reads a line from *is* and stores it into *s*. If a character *delimiter* is specified, then getline() will use *delimiter* to decide when to stop reading data.

For example, the following code reads a line of text from **stdin** and displays it to **stdout**:

```
string s;
getline( cin, s );
cout << "You entered " << s << endl;
```

After getting a line of data in a string, you may find that string streams are useful in extracting data from that string. For example, the following code reads numbers from standard input, ignoring any "commented" lines that begin with double slashes:

```
// expects either space-delimited numbers or lines that start with
// two forward slashes (//)
string s;
while( getline(cin,s) )
{
  if( s.size() >= 2 && s[0] == '/' && s[1] == '/' )
  {
    cout << "  ignoring comment: " << s << endl;
  }
  else
  {
    istringstream ss(s);
    double d;
    while( ss >> d )
    {
      cout << "  got a number: " << d << endl;
    }
  }
}
```

When run with a user supplying input, the above code might produce this output:

```
// test
  ignoring comment: // test
23.3 -1 3.14159
  got a number: 23.3
  got a number: -1
  got a number: 3.14159
// next batch
  ignoring comment: // next batch
1 2 3 4 5
  got a number: 1
  got a number: 2
  got a number: 3
  got a number: 4
  got a number: 5
50
  got a number: 50
```

C++ String function: insert

Syntax

```
#include <string>
iterator insert( iterator i, const char& ch );
string& insert( size_type index, const string& str );
string& insert( size_type index, const char* str );
string& insert( size_type index1, const string& str, size_type index2, size_type num );
string& insert( size_type index, const char* str, size_type num );
string& insert( size_type index, size_type num, char ch );
void insert( iterator i, size_type num, const char& ch );
void insert( iterator i, iterator start, iterator end );
```

The very multi-purpose insert() function either:

- inserts *ch* before the character denoted by *i*,
- inserts *str* into the current string, at location *index*,
- inserts a substring of *str* (starting at *index2* and *num* characters long) into the current string, at location *index1*,
- inserts *num* characters of *str* into the current string, at location *index*,
- inserts *num* copies of *ch* into the current string, at location *index*,
- inserts *num* copies of *ch* into the current string, before the character denoted by *i*,
- or inserts the characters denoted by *start* and *end* into the current string, before the character specified by *i*.

C++ String function: length

Syntax

```
#include <string>
size_type length() const;
```

The length() function returns the number of elements in the current string, performing the same role as the size() function.

C++ String function: max_size

Syntax

```
#include <string>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the string can hold. The max_size() function should not be confused with the size() or capacity() functions, which return the number of elements currently in the string and the the number of elements that the string will be able to hold before more memory will have to be allocated, respectively.

C++ String function: push_back

Syntax

```
#include <string>
void push_back( const TYPE& val );
```

The push_back() function appends *val* to the end of the string.

For example, the following code puts 10 integers into a list:

```
list<int> the_list;
for( int i = 0; i < 10; i++ )
  the_list.push_back( i );
```

When displayed, the resulting list would look like this:

```
0 1 2 3 4 5 6 7 8 9
```

push_back() runs in constant time.

C++ String function: rbegin

Syntax

```
#include <string>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current string.

rbegin() runs in constant time.

C++ String function: rend

Syntax

```
#include <string>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current string.

rend() runs in constant time.

C++ String function: replace

Syntax

```
  #include <string>
  string& replace( size_type index, size_type num, const string& str );
  string& replace( size_type index1, size_type num1, const string& str, size_type index2,
size_type num2 );
  string& replace( size_type index, size_type num, const char* str );
  string& replace( size_type index, size_type num1, const char* str, size_type num2 );
  string& replace( size_type index, size_type num1, size_type num2, char ch );
  string& replace( iterator start, iterator end, const string& str );
  string& replace( iterator start, iterator end, const char* str );
  string& replace( iterator start, iterator end, const char* str, size_type num );
  string& replace( iterator start, iterator end, size_type num, char ch );
```

The function replace() either:

- replaces characters of the current string with up to *num* characters from *str*, beginning at *index*,
- replaces up to *num1* characters of the current string (starting at *index1*) with up to *num2* characters from *str* beginning at *index2*,
- replaces up to *num* characters of the current string with characters from *str*, beginning at *index* in *str*,
- replaces up to *num1* characters in the current string (beginning at *index1*) with *num2* characters from *str* beginning at *index2*,
- replaces up to *num1* characters in the current string (beginning at *index*) with *num2* copies of *ch*,
- replaces the characters in the current string from *start* to *end* with *str*,
- replaces characters in the current string from *start* to *end* with *num* characters from *str*,
- or replaces the characters in the current string from *start* to *end* with *num* copies of *ch*.

For example, the following code displays the string "They say he carved it himself...find your soul-mate, Homer."

```
  string s = "They say he carved it himself...from a BIGGER spoon";
  string s2 = "find your soul-mate, Homer.";
  s.replace( 32, s2.length(), s2 );
  cout << s << endl;
```

C++ String function: reserve

Syntax

```
  #include <string>
  void reserve( size_type size );
```

The reserve() function sets the capacity of the string to at least *size*.

reserve() runs in linear time.

C++ String function: resize

Syntax

```
#include <string>
void resize( size_type size, const TYPE& val = TYPE() );
```

The function resize() changes the size of the string to *size*. If *val* is specified then any newly-created elements will be initialized to have a value of *val*.

This function runs in linear time.

C++ String function: rfind

Syntax

```
#include <string>
size_type rfind( const string& str, size_type index );
size_type rfind( const char* str, size_type index );
size_type rfind( const char* str, size_type index, size_type num );
size_type rfind( char ch, size_type index );
```

The rfind() function either:

- returns the location of the first occurrence of *str* in the current string, doing a reverse search from *index*, string::npos if nothing is found,
- returns the location of the first occurrence of *str* in the current string, doing a reverse search from *index*, searching at most *num* characters, string::npos if nothing is found,
- or returns the location of the first occurrence of *ch* in the current string, doing a reverse search from *index*, string::npos if nothing is found.

For example, in the following code, the first call to rfind() returns string::npos, because the target word is not within the first 8 characters of the string. However, the second call returns 9, because the target word is within 20 characters of the beginning of the string.

```
int loc;
string s = "My cat's breath smells like cat food.";
loc = s.rfind( "breath", 8 );
cout << "The word breath is at index " << loc << endl;
loc = s.rfind( "breath", 20 );
cout << "The word breath is at index " << loc << endl;
```

C++ String function: size

Syntax

```
#include <string>
size_type size() const;
```

The size() function returns the number of elements in the current string.

C++ String function: substr

Syntax

```
#include <string>
string substr( size_type index, size_type length = npos );
```

The substr() function returns a substring of the current string, starting at *index*, and *length* characters long. If *length* is omitted, it will default to string::npos, and the substr() function will simply return the remainder of the string starting at *index*.

For example:

```
string s("What we have here is a failure to communicate");
string sub = s.substr(21);
cout << "The original string is " << s << endl;
cout << "The substring is " << sub << endl;
```

displays

```
The original string is What we have here is a failure to communicate
The substring is a failure to communicate
```

C++ String function: swap

Syntax

```
#include <string>
void swap( container& from );
```

The swap() function exchanges the elements of the current string with those of *from*. This function operates in constant time.

For example, the following code uses the swap() function to exchange the values of two strings:

```
string first( "This comes first" );
string second( "And this is second" );
first.swap( second );
cout << first << endl;
cout << second << endl;
```

The above code displays:

```
And this is second
This comes first
```

C++ String Stream Functions

String streams are similar to the <iostream> and <fstream> libraries, except that string streams allow you to perform I/O on strings instead of streams. The <sstream> library provides functionality similar to sscanf() and sprintf() in the standard C library. Three main classes are available in <sstream>:

- stringstream - allows input and output
- istringstream - allows input only
- ostringstream - allows output only

String streams are actually subclasses of iostreams, so **all of the functions available for iostreams are also available for stringstream**. See the C++ I/O functions for more information.

rdbuf	get the buffer for a string stream
str	get or set the stream's string

String Stream Constructors

Syntax

```
#include <sstream>
stringstream()
stringstream( openmode mode )
stringstream( string s, openmode mode )
ostringstream()
ostringstream( openmode mode )
ostringstream( string s, openmode mode )
istringstream()
istringstream( openmode mode )
istringstream( string s, openmode mode )
```

The stringstream, ostringstream, and istringstream objects are used for input and output to a string. They behave in a manner similar to fstream, ofstream and ifstream objects.

The optional *mode* parameter defines how the file is to be opened, according to the io stream mode flags. An ostringstream object can be used to write to a string. This is similar to the C sprintf() function. For example:

```
ostringstream s1;
int i = 22;
s1 << "Hello " << i << endl;
string s2 = s1.str();
cout << s2;
```

An istringstream object can be used to read from a string. This is similar to the C sscanf() function. For example:

```
istringstream stream1;
string string1 = "25";
stream1.str(string1);
int i;
stream1 >> i;
cout << i << endl;  // displays 25
```

You can also specify the input string in the istringstream constructor as in this example:

```
string string1 = "25";
istringstream stream1(string1);
int i;
stream1 >> i;
cout << i << endl;  // displays 25
```

A stringstream object can be used for both input and output to a string like an fstream object.

String Stream Operators

Syntax

```
#include <sstream>
operator<<
operator>>
```

Like C++ I/O Streams, the simplest way to use string streams is to take advantage of the overloaded << and >> operators.

The << operator inserts data into the stream. For example:

```
stream1 << "hello" << i;
```

This example inserts the string "hello" and the variable *i* into *stream1*. In contrast, the >> operator extracts data out of a string stream:

```
stream1 >> i;
```

This code reads a value from *stream1* and assigns the variable *i* that value.

C++ String stream function: rdbuf

Syntax

```
#include <sstream>
stringbuf* rdbuf();
```

The *rdbuf()* function returns a pointer to the string buffer for the current string stream.

C++ String stream function: str

Syntax

```
#include <sstream>
void str( string s );
string str();
```

The function *str()* can be used in two ways. First, it can be used to get a copy of the string that is being manipulated by the current stream string. This is most useful with output strings. For example:

```
ostringstream stream1;
stream1 << "Testing!" << endl;
cout << stream1.str();
```

Second, *str()* can be used to copy a string into the stream. This is most useful with input strings. For example:

```
istringstream stream1;
string string1 = "25";
stream1.str(string1);
```

str(), along with *clear()*, is also handy when you need to clear the stream so that it can be reused:

```
istringstream stream1;
float num;

// use it once
string string1 = "25 1 3.235\n1111111\n222222";
stream1.str(string1);
while( stream1 >> num ) cout << "num: " << num << endl;  // displays numbers, one per line

// use the same string stream again with clear() and str()
string string2 = "1 2 3 4 5  6 7 8 9 10";
stream1.clear();
stream1.str(string2);

while( stream1 >> num ) cout << "num: " << num << endl;  // displays numbers, one per line
```

C++ Miscellaneous Functions

auto_ptr	create pointers that automatically destroy objects

C++ Miscellaneous function: auto_ptr

Syntax

```
#include <memory>
auto_ptr<class TYPE> name
```

The auto_ptr class allows the programmer to create pointers that point to other objects. When auto_ptr pointers are destroyed, the objects to which they point are also destroyed.

The auto_ptr class supports normal pointer operations like =, *, and ->, as well as two functions TYPE* get() and TYPE* release(). The get() function returns a pointer to the object that the auto_ptr points to. The release() function acts similarly to the get() function, but also relieves the auto_ptr of its memory destruction duties. When an auto_ptr that has been released goes out of scope, it will not call the destructor of the object that it points to.

Warning: It is generally a **bad idea** to put auto_ptr objects inside C++ STL containers. C++ containers can do funny things with the data inside them, including frequent reallocation (when being copied, for instance). Since calling the destructor of an auto_ptr object will free up the memory associated with that object, any C++ container reallocation will cause any auto_ptr objects to become invalid.

Example

```
#include <memory>
using namespace std;

class MyClass
{
public:
  MyClass() {} // nothing
  ~MyClass() {} // nothing
  void myFunc() {} // nothing
};

int main()
{
  auto_ptr<MyClass> ptr1(new MyClass), ptr2;

  ptr2 = ptr1;
  ptr2->myFunc();

  MyClass* ptr = ptr2.get();

  ptr->myFunc();

  return 0;
}
```

Revision History

Date	Version No.	Revision
24-Jan-2008	1.0	Initial release
19-May-2008	2.0	Processor specific C keywords and intrinsic functions added
07-Nov-2008	3.0	Added and changed processor specific keywords
06-Apr-2009	4.0	Added C++