Learning C++

Anzong Zheng

January 20, 2015

Contents

Chapter 1

Naming Convention

```
1. member variable: m_varName.
2. function name: FunName().
3. function name with abbreviation: IOException and GuiElement and HttpServer
4. class name: ClassName
5. enum: kEnumType
6. HaveFlag(), IsEmpty()
```

Chapter 2

Macros

Macros for different systems:

```
1. __unix__
2. __linux__
3. __APPLE__
4. _WIN32
```

Windows specific:

```
1. __LINE__: Line number in current source file.
2. __FILE__: file name of current source file.
3. __DATE__: compile date, like: Aug 28 2011.
4. __TIME__: compile time, like 06:43:59.
5. __STDC__: compatible with ANSI/ISO C standards, work with #if.
6. __TIMESTAMP__: the last time modified the current file.
7. __cplusplus: compile as c plus plus not c, work with #ifdef.
```

Chapter 3

C++ Types

3.1 size_t

Unsigned integral type

It is a type able to represent the size of any object in bytes: **size_t** is the type of returned by the **sizeof** operator and is widely used in the standard library to represent sizes and counts.

In **<cstring >**, it is used as the type of the parameter *num* in the functions **memchr, memcmp, memcpy, memmove, memset, strncat, strncpy,** and **strxfrm**, which in all cases it is used to specify the maximum number of bytes or characters the function has to affect.

It is also used as the return type for **strcspn, strlen, strspn** and **strxfrm** to return sizes and lengths.

3.2 Differences between main char* and _tmain, _TCHAR*

_tmain does not exist in C++. **main** does.

_tmain is a Microsoft extension.

main is, according to the C++ standard, the program's entry point. It has one of these two signatures:

int main();
int main(int argc, char* argv[]);

Microsoft has added a **wmain** which replaces the second signature with this:
int wmain(int argc, wchar_t* argv[]);

And then, to make it easier to switch between Unicode (UTF-16) and their multibyte character set, they've defined **_tmain** which, if Unicode is enabled, is compiled as **wmain**, and otherwise as **main**.

As for the second part of your question, the first part of the puzzle is that your main function is wrong. **wmain** should take a **wchar_t** argument, not **char**. Since the compiler doesn't enforce this for the **main** function, you get a program where an array of **wchar_t** strings are passed to the **main** function, which interprets them as **char** strings.

Now, in UTF-16, the character set used by Windows when Unicode is enabled, all the ASCII characters are represented as the pair of bytes
0 followed by the ASCII value.

And since the x86 CPU is little-endian, the order of these bytes are swapped, so that the ASCII value comes first, then followed by a null byte.

And in a char string, how is the string usually terminated? Yep, by a null byte. So your program sees a bunch of strings, each one byte long.

In general, you have three options when doing Windows programming:

- Explicitly use Unicode (call **wmain**, and for every Windows API function which takes char-related arguments, call the **-W** version of the function. Instead of CreateWindow, call CreateWindowW). And instead of using **char** use **wchar_t**, and so on
- Explicitly disable Unicode. Call main, and CreateWindowA, and use **char for strings**.
- Allow both. (call **_tmain**, and CreateWindow, which resolve to main/**_tmain** and CreateWindowA/CreateWindowW), and use TCHAR instead of char/wchar_t.

The same applies to the string types defined by windows.h: LPCTSTR resolves to either LPCSTR or LPCWSTR, and for every other type that includes char or wchar_t, a -T- version always exists which can be used instead.

Note that all of this is Microsoft specific. TCHAR is not a standard C++ type, it is a macro defined in windows.h. **wmain** and **_tmain** are also defined by Microsoft only.

3.3 What are TCHAR, WCHAR, LPSTR, LPWSTR, LPCSTR etc.?

Many C++ Windows programmers get confused over what bizarre identifiers like **TCHAR, LPCSTR** are. In this article, I would attempt by test to clear out the fog.

In general, a character can be represented in 1 byte or 2 bytes. Let's say 1-byte character is **ANSI character** -all English characters are represented through this *encoding*. And let's say a 2-byte character is **Unicode**, which can represent All languages in the world.

the Visual C++ compiler supports **char** and **wchar_t** as native data-types for **ANSI** and **Unicode** characters, respectively. Though there is more concrete definition of *Unicode*, but for understanding assume it as two-byte character which Windows OS uses for multiple language support.

There is more Unicode than 2-bytes character representation Windows uses. Microsoft Windows us e UTF-16 charachter encoding.

what if you want your C/C++ code to be independent of encoding/mode used?

Suggestion: Use generic data-types and names to represent characters and string.
For example, instead of replacing:

```
char cResponse; // 'Y' or 'N'
char sUsername[64];
// str* functions
```

with

```
wchar_t cResponse; // 'Y' or 'N'
wchar_t sUsername[64];
// wcs* functions
```

In order to support multi-lingual (i.e., Unicode) in your language, you can simply code it in more generic manner:

```
# include<TCHAR.H>// Implicit or explicit include
TCHAR cResponse; // 'Y' or 'N'
TCHAR sUsername[64];
//_tcs* functioins
```

The following project setting in General page describes which Character Set is to be used for compilation: (*General ->Character Set*)

This way, when your project is being compiled as Unicode, the **TCHAR** would translate to **wchar_t** If it is being compiled as ANSI/MBCS, it would be translated to **char**. you are free to use **char and wchar_t**, and project settings will not affect any direct use of these keywords.

TCHAR is defined as:

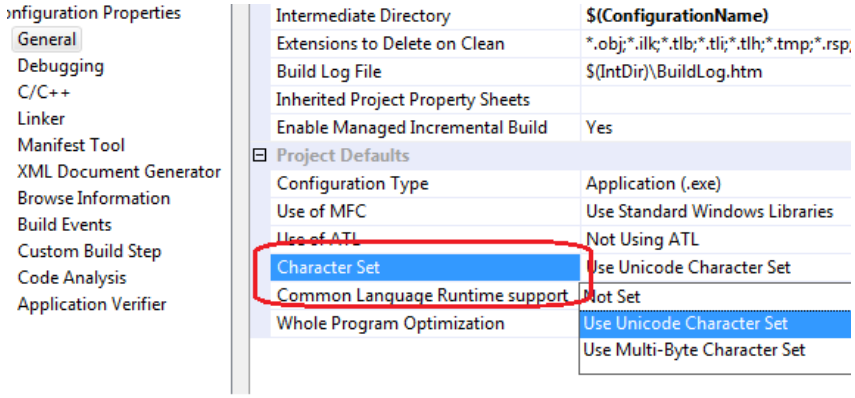

Figure 3.1: Character setting

```
#ifdef _UNICODE
typedef wchar_t TCHAR;
#else
typedef char TCHAR;
#endif
```

The macro_UNICODE is defined when you set Character Set to *"Use Unicode Character Set"*, and therefore **THCAR** would mean **wchar_t**. When Character Set if set to *"Use Multi-Byte Character Set"*, TCHAR would mean **char**.

Likewise, to support multiple character-set using single code base, and possible supporting multi-language, use specific functions(macros). Instead of using **strcpy, strlen, strcat** (including the secure versions suffixed with_s); or **wcscpy, wcslen, wcscat** (including secure), you should better use **_tcscpy, _tcslen, _tcscat** functions.

As you know **strlen** is prototyped as:

```
size_t strlen(const char*);
```

And, **wcslen** is prototyped as:

```
size_t wcslen(const wchar_t*);
```

You may better use **_tcslen**, which is *logically* prototyped as:

```
size_t _tcslen(const TCHAR*);
```

WC is for Wide Character. Therefore, **wcs** turns to be wide-character-string. This way, **_tcs** would mean **_T** Character String. And you know **_T** may be **char** or **what_t**, logically.

But, in reality, **_tcslen** (and other **_tch** functions) are **not** functions, but **macros**. They are defined simply as:

```
#ifdef _UNICODE
#define _tcslen wcslen
#else
#define _tchlen strlen
#endif
```

You should refer **TCHAR.H** to lookup more macro definitions like this.

You might ask why they are defined as macros, and not implemented as functions instead? The reason is simple: A library or DLL may export a single function, with same name and prototype (Ignore overloading concept of C++). For instance, when you export a function as:

```
void _TPrintChar(char);
```

How the client is supposed to call it as?

```
void _TPrintChar(wchar_t);
```

_TPrintChar cannot be magically converted into function taking 2-byte character. There has to be two separate functions:

```
void PrintCharA(char); // A = ANSI
void PrintCharw(wchar_t); // W= Wide character
```

And a simple macro, as defined below, would hide the difference:

```
#ifdef _UNICODE
void _TPrintChar(wchar _t);
#else
void _TPrintChar(char);
#endif
```

The client would simply call it as:

```
TCHAR cChar;
_TPrintChar(cChar);
```

Note that both **TCHAR** and **_TPrintChar** would map to either Unicode or ANSI, and therefore **cChar** and the argument to function would be either char or **wchar_t**.

Macros do avoid these complications, and allows us to use either ANSI or Unicode function for characters and strings. Most of the Windows functions, that take string or a character are implemented this way, and for programmers convenience, only one function (a macro!) is good. SetWindowText is one example:

```
// winUser.H
#ifdef UNICODE
#define SetWindowText SetWindowTextW
#else
#define SetWindowText SetWindowTextA
#endif // !UNICODE
```

There are very few functions that do not have macros, and are available only with suffixed **W** or **A**. One example is **ReadDirectoryChangesW**, which doesn't have ANSI equivalent.

You all know that we use double quotation marks to represent strings. The string represented in this manner is ANSI-string, having 1-byte each character. Example:

```
"This is ANSI String. Each letter takes 1 byte."
```

The string text given above is **not** Unicode, and would be quantifiable for multi-language support. To represent Unicode string, you need to use prefix **L**. An example:

```
L"This is Unicode string. Each letter would take 2 bytes, including spaces."
```

Note the **L** at the beginning of string, which makes it a Unicode string. All characters (I repeat **all** characters) would take two bytes, including all English letters, spaces, digits, and the null character. Therefore, length of Unicode string would always be in multiple of 2-bytes. A Unicode string of length 7 characters would need 14 bytes, and so on. Unicode string taking 15 bytes, for example, would not be valid in any context.

In general, string would be in multiple of **sizeof(TCHAR)** bytes!

When you need to express hard-coded string, you can use:

```
"ANSI String"; // ANSI
L"Unicode String"; // Unicode

_T("Either string, depending on compilation"); // ANSI or Unicode
// or use TEXT macro, if you need more readability
```

The non-prefixed string is ANSI string, the **L** prefixed string is Unicode, and string specified in **_T** or **TEXT** would be either, depending on compilation. Again, **_T** and **TEXT** are nothing but macros, and are defined as:

```
// SIMPLIFIED
#ifdef _UNICODE
#define _T(c) L##c
#define TEXT(c) L##c
#else
#define _T(c) c
#define TEXT(c) c
#endif
```

The ## symbol is token pasting operator, which would turn **_T("Unicode")** into **L"Unicode"**, where the string passed is argument to macro - If **_UNICODE** is defined. If **_UNICODE** is not defined, **_T("Unicode")** would simply mean "**Unicode**". The token pasting operator did exist even in C language, and is not specific about VC++ or character encoding.

Note that these macros can be used for strings as well as characters. **_T('R')** would turn into **L'R'** or simple **'R'** - former is Unicode character, latter is ANSI character.

No, you cannot use these macros to convert variables (string or character) into Unicode/non-Unicode text. Following is not valid:

```
char c = 'C';
char str[16] = "CodeProject";

_T(c);
_T(str);
```

The bold lines would get successfully compiled in ANSI (Multi-Byte) build, since **_T(x)** would simply be x, and therefore **_T(c)** and **_T(str)** would come out to be **c** and **str**, respectively. But, when you build it with Unicode character set, it would fail to compile:

fcolorboxyellowyellow
```
error C2065: 'Lc': undeclared identifier
error C2065: 'Lstr': undeclared identifer
```

I would not like to insult your intelligence by describing why and what those errors are.

There exist set of conversion routine to convert MBCS to Unicode and vice versa, which I would explain soon.

It is important to note that almost all functions that take string (or character), primarily in Windows API, would have generalized prototype in MSDN and elsewhere. The function **SetWindowTextA/W**, for instance, be classified as:

```
BOOL SetWindowText(HWND, const TCHAR*);
```

But, as you know, **SetWindowText** is just a macro, and depending on your build settings, it would mean either of following:

```
BOOL SetWindowTextA(HWND, const char*);
BOOL SetWindowTextW(HWND, const wchar_t*);
```

Therefore, don't be puzzled if following call fails to get address of this function!

```
HMODULE hDLLHandle;
FARPROC pFuncPtr;

hDLLHandle = LoadLibrary(L"user32.dll");

pFuncPtr = GetProcAddress(hDLLHandle, "SetWindowText");
//pFuncPtr will be null, since there doesn't exist any function with name SetWindowText !
```

From **User32.DLL**, the two functions **SetWindowTextA** and **SetWindowTextW** are exported, not the function with generalized name.

Interestingly, .NET Framework is smart enough to locate function from DLL with generalized name:

```
[DllImport("user32.dll")]
extern public static int SetWindowText(IntPtr hWnd, string lpString);
```

No rocket science, just bunch of ifs and else around **GetProcAddress**!

All of the functions that have ANSI and Unicode versions, would have actual implementation only in Unicode version. That means, when you call SetWindowTextA from your code, passing an ANSI string - it would convert the ANSI string to Unicode text and then would call SetWindowTextW. The actual work (setting the window text/title/caption) will be performed by Unicode version only!

Take another example, which would retrieve the window text, using **GetWindowText**. You call GetWindowTextA, passing ANSI buffer as target buffer. **GetWindowTextA** would first call **GetWindowTextW**, probably allocating a Unicode string (a **wchar_t** array) for it. Then it would convert that Unicode stuff, for you, into ANSI string.

This ANSI to Unicode and vice-versa conversion is not limited to GUI functions, but entire set of Windows API, which do take strings and have two variants. Few examples could be:

- CreateProcess
- GetUserName
- OpenDesktop
- DeleteFile
- etc

It is therefore very much recommended to call the Unicode version directly. In turn, it means you should **always** target for Unicode builds, and not ANSI builds - just because you are accustomed to using ANSI string for years. Yes, you may save and retrieve ANSI strings, for example in file, or send as chat message in your messenger application. The conversion routines do exist for such needs.

Note: There exists another typedef: **WCHAR**, which is equivalent to **wchar_t**.

The **TCHAR** macro is for a single character. You can definitely declare an array of **TCHAR**. What if you would like to express a *character-pointer*, or a *const-character-pointer* - Which one of the following?

```
// ANSI characters
foo_ansi(char*);
foo_ansi(const char*);
/*const*/ char* pString;

// Unicode/wide-string
foo_uni(WCHAR*);
wchar_t* foo_uni(const WCHAR*);
/*const*/ WCHAR* pString;

// Independent
foo_char(TCHAR*);
foo_char(const TCHAR*);
/*const*/ TCHAR* pString;
```

After reading about **TCHAR** stuff, you would definitely select the last one as your choice. There are better alternatives available to represent strings. For that, you just need to include Windows.h. **Note**: If your project implicitly or explicitly includes Windows.h, you need not include TCHAR.H

First, revisit old string functions for better understanding. You know **strlen**:

```
size_t strlen(const char*);
```

Which may be represented as:

```
size_t strlen(LPCSTR);
```

The meaning goes like:

- **LP** - Long Pointer
- **C** - Constant
- **STR** - string

Essentially, **LPCSTR** would mean (Long) Pointer to a Constant String.

Let's represent **strcpy** using new style type-names:

```
LPSTR strcpy(LPSTR szTarget, LPCSTR szSource);
```

The type of **szTarget** is **LPSTR**, without **C** in the type-name. It is defined as:

```
typedef char* LPSTR;
```

Note that the **szSource** is **LPCSTR**, since **strcpy** function will not modify the source buffer, hence the const attribute. The return type is non-constant-string: **LPSTR**.

Alright, these str-functions are for ANSI string manipulation. But we want routines for 2-byte Unicode strings. For the same, the equivalent wide-character str-functions are provided. For example, to calculate length of wide-character (Unicode string), you would use **wcslen**:

```
size_t nLength;
nLength = wcslen(L"Unicode");
```

The prototype of **wcslen** is:

```
size_t wcslen(const wchar_t* szString); // Or WCHAR*
```

And that can be represented as:

```
size_t wcslen(LPCWSTR szString);
```

Where the symbol **LPCWSTR** is defined as:

```
typedef const WCHAR* LPCWSTR;
// const wchar_t*
```

Which can be broken down as:

- **LP** - Long Pointer
- **C** - Constant
- **STR** - string

Similarly, **strcpy** equivalent is **wcscpy**, for Unicode strings:

```
wchar_t* wcscpy(wchar_t* szTarget, const wchar_t* szSource
```

Which can be represented as:

```
LPWSTR wcscpy(LPWSTR szTarget, LPWCSTR szSource);
```

Where the target is non-constant wide-string (**LPWSTR**), and source is constant-wide-string.

There exist set of equivalent wcs-functions for str-functions. The str-functions would be used for plain ANSI strings, and wcs-functions would be used for Unicode strings.

Though, I already advised to use Unicode native functions, instead of ANSI-only or TCHAR-synthesized functions. The reason was simple - your application must only be Unicode, and you should not even care about code portability for ANSI builds. But for the sake of completeness, I am mentioning these generic mappings.

To calculate length of string, you may use **_tcslen** function (a macro). In general, it is prototyped as:

```
size_t _tcslen(const TCHAR* szString);
```

Or, as:

```
size_t _tcslen(LPCTSTR szString);
```

- **LP** - Long Pointer
- **C** - Constant
- **T = TCHAR**
- **STR** - string

Depending on the project settings, **LPCTSTR** would be mapped to either **LPCSTR (ANSI)** or **LPCWSTR (Unicode)**.

Note: strlen, wcslen or **_tcslen** will return number of c**haracters** in string, not the number of bytes.

The generalized string-copy routine **_tcscpy** is defined as:

```
size_t _tcscpy(TCHAR* pTarget, const TCHAR* pSource);
```

Or, in more generalized form, as:

```
size_t _tcscpy(LPTSTR pTarget, LPCTSTR pSource);
```

You can deduce the meaning of **LPTSTR**!

Usage Examples First, a broken code:

```
int main()

TCHAR name[] = "Saturn";
int nLen; // Or size_t

lLen = strlen(name);
```

On ANSI build, this code will successfully compile since **TCHAR** would be char, and hence name would be an array of char. Calling **strlen** against name variable would also work flawlessly.

Alright. Let's compile the same with with **UNICODE/_UNICODE** defined (i.e. "*Use Unicode Character Set*" in project settings). Now, the compiler would report set of errors:

- error C2440: 'initializing' : cannot convert from 'const char [7]' to 'TCHAR []'
- error C2664: 'strlen' : cannot convert parameter 1 from 'TCHAR []' to 'const char *'

And the programmers would start committing mistakes by correcting it this way (first error):

```
TCHAR name[] = (TCHAR*)"Saturn";
```

Which will not pacify the compiler, since the conversion is not possible from **TCHAR*** to **TCHAR[7]**. The same error would also come when native ANSI string is passed to a Unicode function:

```
nLen = wcslen("Saturn");
// ERROR: cannot convert parameter 1 from 'const char [7]' to 'const wchar_t *'
```

And you'd think you've attained one more experience level in pointers! You are wrong - the code would give incorrect result, and in most cases would simply cause Access Violation. Typecasting this way is like passing a float variable where a structure of 80 bytes is expected (logically).

The string "**Saturn**" is sequence of 7 bytes:

'S' (83)	textbf'a' (97)	textbf't' (116)	textbf'u' (117)	textbf'r' (114)	textbf'n' (110)	textbf'\0' (0)

But when you pass same set of bytes to wcslen, it treats each 2-byte as a single character. Therefore first two bytes [97, 83] would be treated as one character having value: 24915 (97¡¡8 — 83). It is Unicode character: ?. And the next character is represented by [117, 116] and so on.

For sure, you didn't pass those set of Chinese characters, but improper typecasting has done it! Therefore it is very essential to know that type-casting **will not** work! So, for the first line of initialization, you must do:

```
TCHAR name[] = _T("Saturn");
```

Which would translate to 7-bytes or 14-bytes, depending on compilation. The call to wcslen should be:

```
wcslen(L"Saturn");
```

In the sample program code given above, I used strlen, which causes error when building in Unicode. The non-working solution is C-sytle typecast:

```
lLen = strlen ((const char*)name);
```

On Unicode build, name would be of 14-bytes (7 Unicode characters, including null). Since string "**Saturn**" contains only English letters, which can be represented using original ASCII, the Unicode letter 'S' would be represented as [83, 0]. Other ASCII characters would be represented with a zero next to them. Note that 'S' is now represented as **2-byte** value 83. The end of string would be represented by **two bytes** having value 0.

So, when you pass such string to strlen, the first character (i.e. first byte) would be correct ('S' in case of "Saturn"). But the second character/byte would indicate end of string. Therefore, strlen would return incorrect value 1 as the length of string.

As you know, Unicode string may contain non-English characters, the result of strlen would be more undefined.

In short, typecasting will not work. You either need to represent strings in correct form itself, or use ANSI to Unicode, and vice-versa, routines for conversions.

(There is more to add from this location, stay tuned!)

Now, I hope you understand the following signatures:

```
BOOL SetCurrentDirectory( LPCTSTR lpPathName );
DWORD GetCurrentDirectory(DWORD nBufferLength,LPTSTR lpBuffer);
```

Continuing. You must have seen some functions/methods asking you to pass **number of characters**, or returning the number of characters. Well, like GetCurrentDirectory, you need to pass number of characters, and **not** number of bytes. For example:

```
TCHAR sCurrentDir[255];

// Pass 255 and not 255*2
GetCurrentDirectory(sCurrentDir, 255);
```

On the other side, if you need to allocate number or characters, you must allocate proper number of bytes. In C++, you can simply use new:

```
LPTSTR pBuffer; // TCHAR*

pBuffer = new TCHAR[128]; // Allocates 128 or 256 BYTES, depending on compilation.
```

But if you use memory allocation functions like malloc, LocalAlloc, GlobalAlloc, etc; you must specify the number of bytes!

```
pBuffer = (TCHAR*) malloc (128 * sizeof(TCHAR) );
```

cTypecasting the return value is required, as you know. The expression in malloc's argument ensures that it allocates desired number of bytes - and makes up room for desired number of characters.

3.4 typedef and struct define

typedef is a keyword in the C and C++ programming languages. The purpose of **typedef** is to form complex types for more-basic machine types and assign simpler names to such combinations. They are most often used when a standard declaration is cumbersome, potentially confusing, or likely to vary from one implementation to another.

A **typedef** can be used to simplify the declaration for a compound type (struct, union) or pointer type. For example:

```
struct MyStruct
{
int data1;
char data2;
}
```

Here (above) a **struct MyStruct** data type has been defined. To declare a variable of this in C (below) the **struct** key word is required (though it may be omitted in C++).

The defined struct can be now be used to generate variables, such as:

```
struct MyStruct a;
```

A **typedef** can be used to eliminate the need for **struct** key word in C. For example, with:

```
typedef struct Mystruct newtype;
```

We can now create a variable of this type with:

```
newtype a;
```

Note that the structure definition and typedef can instead be compiled into a single statement:

```
tepydef struct Mystruct
{
int data1;
char data2;
} newtype;
```

Another example for vertices definition is as bellow:

```
/* the usual 3-space position of a vertex */
typedef struct Vertex
{
float x, y, z;
} Vertices;
```

Note that the **Vertices** equal to **struct Vertex**. Its function likes **int, double...**

Another frequently used form of **typedef** is:

```
typedef char field [50];
```

which means field type is char[50].

3.5 Type conversion: atoi(), scanf() (replace itoa())

3.5.1 atoi()

Convert string to integer

```
int atoi (const char *str);
```

atoi parses the **C-string** *str* interpreting its content as an integral number, which is returned as a value of **type int**.

The function first discards as many **whitespace characters** (as in **isspace**) as necessary until the **first non-whitespace character** is found. Then, starting from this character, takes an optional initial *plus* or *minus* sign followed by as many **base-10 digits** as possible, and interprets them as a numerical value.

The string can contain additional characters after those form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in *str* is not a valid integral number, or if no such sequence exists because either *str* is empty or it contains only whitespace only whitespace characters, no conversion is performed and **zero** is returned.

EXAMPLE

```
/* atoi example*/
#include <stdio.h> /* printf, fgets */
#include <stdlib.h> /* atoi */

int main()
{
int i;
char buffer[256];
printf ("Enter a number: ");
fgets (buffer, 256, stdin);
i = atoi (buffer);
printf ("The value entered is %d. Its double is %d.\n",i,i*2);
return 0;
}
```

3.5.2 itoa()

NOTE: As **itoa()** is indeed **non-standard**, as mentioned by several helpful commenters, it is best to use **sprintf(target_string, "%d", source_int)** or (better because it's safe from buffer overflows) **snprintf(target_string, size_of_target_string_in_bytes, "%d", source_int)**. **Convert integer to string**

```
char *  itoa ( int value, char * str, int base );
```

Converts an integer value to a **null-terminated string** using the specified base and stores the result in the array given by *str* parameter.

If *base* is 10 and value is negative, the resulting string is preceded with a minus sign (-). With any other base, value is always considered unsigned.

str should be an array long enough to contain any possible value: **(sizeof(int)*8+1)** for **radix=2, i.e. 17** bytes in 16-bits platforms and 33 in 32-bits platforms.

EXAMPLE

```
/* atoi example*/
/* itoa example */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
int i;
char buffer [33];
printf ("Enter a number: ");
scanf ("%d",&i);
itoa (i,buffer,10);
printf ("decimal: %s\n",buffer);
itoa (i,buffer,16);
printf ("hexadecimal: %s\n",buffer);
itoa (i,buffer,2);
printf ("binary: %s\n",buffer);
return 0;
}
```

3.5.3 sprintf()

```
int sprintf( char* buffer, const char* format, ... );
```

EXAMPLE

```
void PrettyFormat( int i, char* buf )
{
// Here's the code, neat and simple:
sprintf( buf, "%d", i );
}
```

NOTE: buffer size needs to be enough for containing integer i.

Chapter 4

C++ Settings

4.1 Compile C++ file in Visual studio

The problem is, Visual studio doesn't really know, what to do with your .cpp file. Is it a program? Try the following:

- **File — New project**
- **Visual C++ — Win32 — Win32 Project**
- Select a name and location for the project
- Next
- Choose **Console application**
- Choose **Deselect Precompiled header**
- (optionally) Deselect **SDL checks**
- Finish
- Right-click on **Source files** and choose **Add — New Item...**
- choose **C++ File**
- Choose name for this file
- Write the following inside:

```
#include <sdtio.h>

int main(int argc, char * argv[])
{
printf("Hello, world!\n");
return 0;
}
```

If precompile headers, it should write:

```
#include <sdtio.h>

int _tmain(int argc, _TCHAR* argv[])
{
printf("Hello, world!\n");
return 0;
}
```

4.2 Stop C++ console application from existing immediately

From http://stackoverflow.com/questions/2529617/how-to-stop-c-console-application-from-exiting-immed

At the end of main function, call std::getchar(). This will get a single character from stdin, thus giving the ”press any key to continue” sort of behavior (if you actually want a ”press any key” message, you’ll have to print one yourself).

#include ¡stdio¿ need to be included for getchar

```
#include "stdafx.h"
#include <cstdio>  // getchar().

int equal_strings(char *s1, char *s2);

int _tmain(int argc, _TCHAR* argv[])
{
printf("Hello world!\n");
std::getchar();

return 0;
}
```

4.3 Debugging Multiple Programs

4.3.1 Introduction

With the visual Studio debugger, you can debug programs running in multiple processes. You can think of a process as an instance of a program running in its own memory space with its own object code, data, and resources. When you start a program by launching an EXE, for example, the system scheduler creates a new process for that program. If you launch multiple instances of the program, it creates multiple processes. The operating system creates other processes automatically, for its own purpose.

4.3.2 Techniques for Debugging Multiple Process

In Visual Studio .NET, you can debug multiple processes within a **Visual Studio solution**. In this case, each process is created by **a separate project** within the solution, so you can think of this as debugging multiple projects. You can do this by **setting multiple startup projects**, or you can **start debugging one project and then start additional projects from Solution Explorer**.

4.3.3 Starting Additional projects

to start one project when another is already running, both projects must be **in the same solution**. You can use Solution Explorer to start the additional project or projects:

To start a project in Solution Explorer

1. In Solution explorer,select the project you want to debugging.
2. Right-click the project name or icon.
3. From the shortcut menu, choose **Debug** and click **Start new instance**.

4.3.4 Switching Between Running Projects

When you are debugging two or more projects in a solution, you can switch between them in either of two ways:

To switch between projects while debugging

- On the **Debug Location** toolbar, select the program you want to switch to from the **Program** list box.

To display the Debug Location toolbar

1. From the **Tools** menu, choose **Customize**.
2. In the **Property** dialog box, choose the **Toolbars** tab and select **Location**.
3. Click **OK**.

Switching to a project makes it the current process for debugging purposes. Any command you select from the **Debug** menu (such as **Break** or **Continue**) will affect the current process as well as the other processes running under control of the debugger.

4.3.5 Stopping the Current Process

To stop the current peocess only

1. From the **Tools** menu, choose **Options**.
2. In the **Options** dialog box, open the **Debugging** folder and choose the **General** category.
3. Select **In break mode, only stop execution of the current process**.

Chapter 5

String and Encoding

5.1 Compare two strings

```
/*****************************************************************************
Compare two strings.  Returns 1 if they are the same, 0 if not.
*****************************************************************************/

int equal_strings(char *s1, char *s2)
{
while (*s1 && *s2)
if (*s1++ != *s2++)
return (0);

if (*s1 != *s2)
return (0);
else
return (1);
}
```

5.2 Difference between std::strlen() and sizeof()

- **strlen()** is used to get the length of an array
- **sizeof()** is used to get the actual size of any type of data in bytes.

Besides, **sizeof()** is a compile-time expression giving you the size of a type or a variable's type. It doesn't care about the value of the variable.

strlen() is a function that takes a pointer to a character, and walks the memory from this character on, looking for a **'\0'** character. It counts the number of characters before it finds the **'\0'** character. In other words, it gives you the length of a C-style **NULL-terminated string**.

The two are almost different. In C++, you do not need either very much, **strlen()** is for C-style strings, which should be replaced by C++-style **std::strings**, whereas the primary application for **sizeof()** in C is an argument to functions like **malloc(), memcpy(),** or **memset()**, all of which you shouldn't use in C++ (use new, **std::copy()**, and **std::fill()** or **constructors**).

NOTE: I think C-string is better than C++-string.

5.3 Difference between <cstring >, <string.h >, <string >

- **<cstring >** will usually import the same things as **<string.h >**, but into the **std** namespace
- **<string.h >** will usually import everything into the global namespace.

NOTE: <string.h > contains C-library string functions. **strlen**(), **strcmp**(), etc. while **<string >** is C++'s header file.

5.4 Split string by a delimiter using vector

The string can be assumed to be composed of words separated by a delimiter. An elegant and efficient way to fulfil this goal:

```
#include <string>
#include <vector>
#include <sstream>

std::vector<std::string> split(const std::string &s, char delim)
{
std::vector<std::string> elems;

std::stringstream ss(s);
std::string item;


while (std::getline(ss, item, delim))
{
if (!item.empty())  /* skipping empty tokens, like "::" using delimiter ':' */
elems.push_back(item);
}

return elems;
}

int main(int argc, char* argv[])
{
std::vector<std::string> x = split("one:two::three", ':');
return 0;
}
```

5.5 Difference between '\0' and "\0"

'\0' is the **null termination character**. It marks the end of the string. Without it, the computer has no way to know how long that group of characters goes. When you print/copy/whatever a string, it just keeps printging/copying chars until it finds that null char... That's when it knows to stop.
Note that when using **double quotes ""** they automatically add the **\0**. So:

```
char cAlphabet[] = "I know all about programming!";
```

is the same as

```
char cAlphabet[] = {'I',' ', 'k','n','o','w',' ','a','l','l',' ','a','b','o','u','t',' ','p','r','o'
```

5.6 Convert from char to wchar_t

Simple example:

```
/* Convert from char to wide-char */
wchar_t* charToWChar(char* text)
{
```

```
    size_t size = strlen(text) + 1;
    wchar_t* wa = new wchar_t[size];

    /* size_t mbstowcs (wchar_t* dest, const char* src, size_t max);
    Convert multibyte string to wide-character string*/
    mbstowcs(wa,text,size);

    return wa;
}
```

5.7 Difference between char * and LPSTR in windows

Chapter 6

Header Settings

6.1 # Include guard

In the C and C++ programming languages, an **# include guard**, sometimes called a **macro guard**, is a particular construct used to avoid the problem of **double inclusion** when dealing with the **include directive**. The addition of **# include guards** to a **header file** is one way to make that file idempotent.

6.1.1 Double inclusion

The following C code demonstrates a real problem that can arise if **# include guards** are missing:

File ”grandfather.h”

```
struct foo
{
int member;
};
```

File ”father.h”

```
#include "grandfather.h"
```

File ”child.c”

```
#include "grandfather.h"
#include "father.h"
```

Here, the file ”child.c” has indirectly included two copies of the text in the **header file ”grandfather.h”**. This causes a compilation error, since the struct type **foo** is apprantly defined twice. In C++, this would be a violation of the **One Definition Rule**.

6.1.2 Use of # include guards

File ”grandfather.h”

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H

struct foo
{
int member;
};

#endif /* GRANDFATHER_H */
```

File ”father.h

```
#include "grandfather.h"
```

File ”child.c”

```
#include "grandfather.h"
#include "father.h"
```

Here, the first inclusion of ”grandfather.h” causes the macro **GRANDFATHER_H** to be defined. Then, when ”child.c” includes ”grandfather.h” the second time, the **# ifndef** test returns false, and the preprocessor skips down to the **# endif**, thus avoiding the second definition of **struct foo**. The program compiles correctly.

Different **naming conventions** for the guard macro may be used by different programmers. Other common forms ot the above include **GRANDFATHER_INCLUDED**, **CREAORSNAME_HHMMSS** (with the appropriate time information substituted) and names generated form a **UUID**. (However, names starting with one or two underscores, such as **_GRANDFATHER_H** and __GRANDFATHER_H, are reserved to the implementation and must not be used by the user.) It is important to avoid duplicating the name in different header files, as including one will prevent the symbols in the other being defined.

6.1.3 Difficulties

In order for # include guards to work properly, each guard must test and conditionally set a different preprocessor macro. Therefore, a project using # include guards must work out a coherent scheme for its include guards, and make sure its scheme doesn't conflict with that of any third-party headers it uses, or with the names of any globally visible macros.

For this reason, most C and C++ implementations provide a non-standard **# pragma once** directive. This directive, inserted at the top of a header file, will ensure that the file is include only once. The **Objective-C** introduced an **# import** directive, which works exactly like **# include**, except that it includes each file only once, thus obviating the need for # include guards.

6.1.4 Using namespaces

Using Namespaces

Section 13.4 in *the programmer's guide to C++* describes how namespaces can be defined, but it does not clearly recommend a way of accessing them. This is now very improtant because of the introduction of the standard library, which is in the **std** namespace. The following example used the **std** namespace, but other namespaces should be used in the same way. In a header file, use the :: operator to access a namespace like this:

```
// A header file called mystuff.h
#include <string>
...
class MyClass{
public:
    std::string address();
    void address( std::string value)
    void dump();
    ...
private:
    std::vector<int> size;
    ...
};
```

Always the :: operator in header files. Never use a **using namespace** statement globally in a header file because it will merge its specified namespace with the global namespace with global namespace in any program that uses the header file. In some files, the :: operator is also the preferred way to access a namespace:

```
// a source code file
#include "MyStuff.h"
...
std::string MyClass::address()
```

```
{
    std::string temp;
    ...
    return temp;
}
```

Building User Defined Namespaces

User defined namespaces are strongly recommended. They simplify the choice and use of names by preventing name clashes. They group related components, such as classes, global functions and constants into logical modules. They greatly assist the safe development of larger programs, which are composed of independent compilation units. Very few of examples in *the programmer's guide to C++* actually use namespaces. This was done to simplify the book's presentation and does not imply that namespaces should not be used in practice.

Building a use defined namespace is easy. Its names should be introduced to the namespace in header files:

```
// file: lottery.h
namespace Lottery {
    const int MAX = 30;

    class Card{
    public:
    Class();
    void generate();
    .....
    };

    void merge (Card c1, Card c2);
}
```

The source files that definition the above include the header files, and use the :: operator to associate with the namespace:

```
// file: lottery.cpp

#include "lottery.h"

Lottery::Card::generate()
{

}

void Lottery::merge(Lottery::Card c1, Lottery::Card c2)
{

}
```

When you use namespaces, it is worth nothing that in at least one leading C++ development environment the class browser is confused by namespaces.

Chapter 7

Programming

7.1 Difference between return 0, return 1, exit(0), EXIT_FAILURE and EXIT_SUCCESS

7.1.1 return value and exit

1 **return from main()**

return from **main()** is equivalent to **exit**. The program terminates immediately execution with **exit status** set as the value passed to **return** or **exit**.

2 **return from an inner function**

return in an inner function (not **main**) will terminate immediately the execution of the specific function returning the given result to the calling function.

3 **exit**

exit from anywhere on your code will terminate execution immediately.

- **Status 0** means the program succeeded.
- **status different than 0** means the program exited due to error or anomaly. If you exit with a status different from 0 you're supposed to print an error message to **stderr** so instead of using **printf** better something like:

  ```
  if(errorOccurred)
  {
  fprintf(stderr, "meaningful message here\n");
  return -1;
  }
  ```

  Note that (depending on the OS you're on) there are some conventions about return codes.
  Also the OS may terminate your program with specific exit status codes if you attempt to do some invalid operations like reading memory you have no access to.

7.1.2 Exit codes C and macros EXIT_SUCCESS and EXIT_FAILURE

The C programming language allows programs exiting or returning from the **main function** to signal success or failure by returning an integer, or returning the **macros EXIT_SUCCESS** and **EXIT_FAILURE**. On Unix-like systems these are equal to 0 and 1 respectively. A C program may also use the **exit()** function specifying or exit macro as the first parameter.

Apart from the macros **EXIT_SUCCESS** and **EXIT_FAILURE**, the C standard does not define the meaning of return codes. Rules for the use of return codes vary on different platforms.
NOTE that the two macros are defined in **<cstdlib>**

7.2 Difference between using namespace std and std::

Using namespace std; is considered **bad practice**. This is not related to performance at all. But consider this: You are using two libraries called Foo and Bar:

```
    using namespace foo;
using namespace bar;
```

Everything works fine, you can call **Blah()** from Foo and **Quux()** from Bar without problems. But one day you upgrade to a new version of Foo 2.0, which now offers a function called **Quux()**. Now you've got a conflict: Both Foo 2.0 and Bar import **Quux()** into your global namespace. This is going to take some effort to fix, especially if the function parameters happen to match.

If you have used **foo::Blah()** and **bar::Quux()** then the introduction of **foo::Quux()** would have been a non-event.

std:: is not harmful at all. It carries very important information (namely "whatever comes after is part of the standard library", and it is still a pretty short and compact prefix). Most of the time, it's no problem at all.

7.3 Difference between . and ->

The difference between **Structure reference .** and **Structure dereference ->** is quite obvious. For example, in a simple structure point:

```
tepedef struct point
{
float a,b;
} point;
```

If we use the definition point a. Then operation **.** should be used to visit members, such as **p1.a**. It means ("member a of object p1").

Otherwise, if the definition point *a is adopted, then operation **->** should be used, such as **p1->a**. It means ("member a of object pointed to by p1")

7.4 Meaning of '*&'

Question: I wrote a function along the lines of this:

```
void myFunc(myStruct *&out)
{
out = new myStruct;
out->field1 = 1;
out->field2 = 2;
}
```

Now in a calling function, I might write something like this:

```
myStruct *data;
myFunc(data);
```

which will fill all the fields in **data**. If I omit the '&' in the declaration, this will not work.

Answer:

The & symbol in a C++ variable declaration means it's a **reference**.

It happens to be a reference to a pointer, which explains the semantics you are seeing; the called function can change the pointer in the calling context, since it has a reference to it.

So, to reiterate, the **operative symbol** here is not ***&**, that combination in itself doesn't mean a whole lot. The * is part of the type **myStruct ***, i.e. "**pointer to myStruct**, and the **&** makes it a reference, so you'd read it as **out is a reference to a pointer to myStruct**.

The original programmer could have helped, in my opinion, by writing it as:

```
void myFunc(myStruct* &out)
```

7.5 Get time and date

C++ shares its date/time functions with C. The tm structure is probably the easiest for a C++ programmer to work with - the following prints today's date:

```
#include <ctime>
#include <iostream>
using namespace std;

int main()
{
time_t t = time(0);   // get time now
struct tm * now = localtime( & t );
cout << (now->tm_year + 1900) << '-'
<< (now->tm_mon + 1) << '-'
<<  now->tm_mday
<< endl;
}
```

7.6 Difference between Continue and Break

Continue jumps straight to the top of the **innermost loop**, where the per-iteration code and continuance check will be carried out.

Break jumps straight to immediately after the **innermost loop** without changing anything.

It may be easier to think of the former jumping to the **closing brace of the innermost loop** while the latter jumps just **beyond** it.

7.7 New and delete in C++

1. New a single space

   ```
   int *a = new int(89);
   delete a;
   ```

2. New a consecutive spaces

   ```
   int *a = new[10];
   delete []a;
   ```

3. New a 2d array spaces A dynamic 2D array is basically an array of pointers to arrays. You should initialize it using a loop:

   ```
   int **a = new int *[sizeX];
   for( int i=0; i<sizeX; i++)
   a[i] = new int [siezeY];
   ```

   The above, for **sizeX = 4** and **sizeY = 5**, would produce the following:

   and then clean up would be:

   ```
   for(int i = 0; i < sizeY; ++i)
   {
   delete [] a[i];
   }
   delete [] a;
   ```

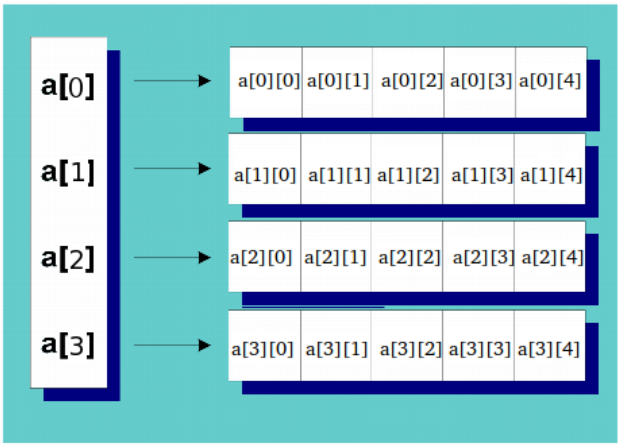

Figure 7.1: New 2D Array

NOTE: as Dietrich Epp pointed out in the comments this is not exactly a light weight solution. An alternative approach would be to use one large block of memory:

```
int *ary = new int[sizeX*sizeY];

// ary[i][j] is then rewritten as
ary[i*sizeY+j]
```

7.8 Use Function Pointer

7.8.1 Basis

A **Function Pointer**, internally, is just the **numerical address** for the code for a function. When a function name is used by itself without parentheses, the vaule is a pointer to the function, just as the name of an array by itself is a pointer to its zeroth element. Function pointers can be stored in **variables, structs, unions, and arrays** and passed to and from functions just like any other type. They can also be called: a variable of type function pointer can be used in place of a function name.

7.8.2 Function pointer declarations

A function pointer declaration looks like a function declaration, except that the function name is wrapped in parentheses and preceded by an asterisk. For example:

```
/* a function taking two int arguments and returning an int */
int function(int x, int y);

/* a pointer to such a function */
int (*pointer)(int x, int y);
```

As with function declarations, the names of the arguments can be omitted.
Here's a short program that uses function pointers:

```
void SimpleExample()
{
/* Step 1: function pointer definition. function for emitting text */
int (*say)(const char *);
```

```
// Step 2: set function puts's address to function pointer say.
say = puts;

// Step 3: pass variables to function pointer say.
say("hello world");
}
```

Note it is convenient to declare a type definition for function pointers like:
typedef int (*func)(int a, int b);

7.8.3 Using Function Pointer inside Struct

In order to make a **struct variable** like **Matrix** to use in ways such as **a.size(), a.zeros()**, we need encapsulate function pointers inside struct.

There are several ways to set function pointer address. One is using the **Default List**:

```
void addMSG(unsigned char *data, int size, struct linkedList *self);

struct linkedList {
int count;
struct msgNode *front;
struct msgNode *back;
void (*addMSG)(unsigned char *, int, struct linkedList *);
} DefaultList = {0, NULL, NULL, addMSG};
```

Another way which I prefer is to initialize in an initialization function:

```
typedef struct pstring_t
{
char * chars;
int (*length)(struct pstring_t * self);
} PString;

int length(PString * self)
{
return std::strlen(self->chars);
}

PString * initializeString(int n)
{
PString *str=(PString*)malloc(sizeof(PString));
str->chars = (char *)malloc(sizeof(char)*n);
str->length = length; // use initialize function to let function pointer length point to function le
return str;
}

int main(int argc, char **argv)
{
PString *p = initializeString(30);
std::strcpy(p->chars,"Hello");
std::printf("\n%d",p->length(p));

return 0;
}
```

7.9 Return a string pointer

Simple Example:

```
char* CreateCommandLineForTetA(char* commandLine, char* fp)
{
    size_t size = std::strlen(commandLine) + std::strlen(fp) + 1;

    char* ret = new char [size];

    std::strcpy(ret, commandLine);
    std::strcat(ret, fp);

    return ret;
}
```

7.10 Reuturn a struct pointer

The variable defined in the function (in ”auto” storage class) will disappear as the function exits, and you’ll return a dangling pointer.

YOu could *accept* a pointer to a **MyStruct** (caller’s responsibility to allocate that) and fill it in; or, you can use **malloc** to create a new one (caller’s responsibility to free it when it’s done). The second option at least lets you keep the function signature you seem to be keen on:

```
MyStruct* myStruct(int num, int size)
{
MyStruct* p = (MyStruct*) malloc(sizeof(MyStruct));
....
return p;
}
```

Chapter 8

Windows Data Types

Link: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa383751(v=vs.85).aspx`

The data types supported by **Windows** are used to define function return values, function and message parameters, and structure members. They define the size and meaning of these elements. For more information about the underlying C/C++ data types, seee **Data Type Ranges**.

The following table contains the following types:character, integer, Boolean, pointer, and handle. The character, integer, and Boolean types are common to most C compilers. Most of the pointer-type begin with a prefix of P or LP. Handles refer to a resource that has been loaded into memory.

For more information about handling 64-bit integers, see **Large Integers**.

Table 8.1: Windows Data Types

Data types	Description
BOOL	A **Boolean** variable (should be **TRUE** or **FALSE**). This type is declared in **WinDef.h** as follows: typedef int BOOL; *Comment: C doesn't have a boolean type.
BOOLEAN	A **Boolean** variable (should be **TRUE** or **FALSE**). This type is declared in **WinNT.h** as follows: typedef BYTE BOOL; *Comment: C doesn't have a boolean type.
BYTE	A **byte (8 bits)** This type is declared in **WinDef.h** as follows: typedef unsigned char BYTE
CALLBACK	The calling convention for callback functions. This type is declared in **WinDef.h** as follows: **#define CALLBACK __stdcall;** **CALLBACK**, **WINAPI**, and **APIENTRY** are all used to define functions with the __stdcall calling convention. Most functions in the Windows API are declared using **WINAPI**. You may wish to use **CALLBACK** for the callback functions that you implement to help identify the function as a callback function.
CCHAR	An 8-bit **Windows (ANSI)** character. This type is declared in **WinNT.h** as follows: typedef char CCHAR;
CHAR	An 8-bit **Windows (ANSI)** character. This type is declared in **WinNT.h** as follows: typedef char CHAR;
CONST	A variable whose value is to remain constant during execution. This type is declared in **WinDef.h** as follows: #define CONST const

Continued on next page

Table 8.1 – continued from previous page	
Data types	Description
DWORD	A 32-bit unsigned integer. The range is 0 through 4294967295 decimal. This type is declared in **IntSafe.h** as follows: typedef unsigned long DWORD;
FLOAT	A floating-point variable. This type is declared in **WinDef.h** as follows: typedef float FLOAT;
HANDLE	A handle to an object. This type is declared in **WinNT.h** as follows: typedef PVOID HANDLE;
HWND	A handle to a **window**. This type is declared in **WinDef.h** as follows: typedef HANDLE HWND;
INT	A 32-bit signed integer. The range is -2147483648 through 2147483647 decimal. This type is declared in **WinDef.h** as follows: typedef int INT;
LONG	A 32-bit signed integer. The range is -2147483648 through 2147483647 decimal. This type is declared in **WinDef.h** as follows: typedef long LONG;
LPCSTR	A pointer to a constant null-terminated string of 8-bit Windows (ANSI) characters. This type is declared in **WinNT.h** as follows: typedef __nullterminated CONST CHAR *LPCSTR;
LPCTSTR	An **LPCWSTR** if **UNICODE** is defined, an **LPCSTR** otherwise. This type is declared in **WinNT.h** as follows: #ifdef UNICODE typedef LPCWSTR LPCTSTR; #else typedef LPCSTR LPCTSTR; #endif
LPCVOID	A pointer to a constant of any type. This type is declared in **WinDef.h** as follows: typedef CONST void *LPCVOID;
LPCWSTR	A pointer to a constant null-terminated string of 16-bit Unicode characters. This type is declared in **WinNT.h** as follows: typedef CONST WCHAR *LPCWSTR;
LPSTR	A pointer to a null-terminated string of 8-bit Windows (ANSI) characters. This type is declared in **WinNT.h** as follows: typedef CHAR *LPSTR;
LPTSTR	An **LPWSTR** if **UNICODE** is defined, an **LPSTR** otherwise. This type is declared in **WinNT.h** as follows: #ifdef UNICODE typedef LPWSTR LPTSTR; #else typedef LPSTR LPTSTR; #endif
SHORT	A 16-bit integer. The range is –32768 through 32767 decimal. This type is declared in **WinNT.h** as follows: typedef short SHORT;

Continued on next page

Table 8.1 – continued from previous page	
Data types	Description
TCHAR	A **WCHAR** if **UNICODE** is defined, a **CHAR** otherwise. This type is declared in **WinNT.h** as follows: #ifdef UNICODE typedef WCHAR TCHAR; #else typedef char TCHAR; #endif
UINT	An unsigned **INT**. The range is 0 through 4294967295 decimal. This type is declared in **WinDef.h** as follows: typedef unsigned int UINT;
VOID	Any type. This type is declared in **WinNT.h** as follows: #define void VOID
WCHAR	A 16-bit Unicode character. This type is declared in **WinNT.h** as follows: typedef wchar_t WCHAR;
WINAPI	The calling convention for system functions. This type is declared in **WinNDef.h** as follows: #define WINAPI __stdcall **CALLBACK**, **WINAPI**, and **APIENTRY** are all used to define functions with the __stdcall calling convention. Most functions in the Windows API are declared using **WINAPI**. You may wish to use **CALLBACK** for the callback functions that you implement to help identify the function as a callback function.
WORD	A 16-bit unsigned integer. The range is 0 through 65535 decimal. This type is declared in **WinDef.h** as follows: typedef unsigned short WORD;

Chapter 9

Casting pointers

There are no rules on casting pointers in C! The language lets you cast any pointer to any other pointer without comment.

But the thing is: there is no data conversion or whatever done! Its solely your own responsibility that the system does not misinterpret the data after the cast - which would generally be the case, leading to runtime error.

So when casting its totally up to you to take care that if data is used from a casted pointer the data is compatible!

C is optimized for performance, so it lacks runtime reflexivity of pointers/references. But that has a price - you as a programmer have to take better care of what you are doing. You have to know on your self if what you want to do is ”legal”.

Chapter 10

Debugging command arguments

Command line arguments can be tested in **Visual Studio** by inputting all parameters in the space left for command arguments. The index of the arguments is 1 because 0 is used to represent the **exe program** itself.

Chapter 11

Multidimentional vector

```
std::vector< std::vector< int > > a; // as Ari pointed
```

Using this for a growing matrix can become complex, as the system will not guarantee that all internal vectors are of the same size. Whenever you grow on the second dimension you will have to explicitly grow all vectors.

```
// grow twice in the first dimension
a.push_back( std::vector<int>() );
a.push_back( std::vector<int>() );

a[0].push_back( 5 ); // a[0].size() == 1, a[1].size() == 0
```

If that is fine with you (it is not really a matrix but a vector of vectors), you should be fine. Else you will need to put extra care to keep the second dimension stable across all the vectors.

If you are planing on a fixed size matrix, then you should consider encapsulating in a class and overriding operator() instead of providing the double array syntax. Read the C++ FAQ regarding this.

Chapter 12

Memory management

12.1 Realloc

http://www.cplusplus.com/reference/cstdlib/realloc/

12.1.1 function realloc <cstdlib>

```
void* realloc (void* ptr, size_t size);
```

12.1.2 Reallocate memory block

Changes the size of the memory block pointer to by *ptr*.

The function may move the mermory block to a new location (whose address is returned by the function).

The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location. If the new *size* is larger, the value of the newly allocated portion in indeterminate.

In case that *ptr* is a null pointer, the function behaves like malloc, assigning a new block of *size* bytes and returning a pointer to its beginning.

12.1.3 C99/C11(C++11)

If *size* is zero, the return value depends on the particular library implementation: it may either be a *null* pointer or some other location that shall not be dereferenced.

If the function fails to allocate the requested block of memory, a null pointer is returned, and the memory block pointed to by argument *ptr* is not deallocated (it is still valid, and with its contents unchanged).

12.1.4 Parameters

ptr Pointer to a memory block previously allocated with malloc, calloc or realloc. Alternatively, this can be a *null pointer*, in which case a new block is allocated (as if malloc was called).

size New size for the memory block, in bytes. `size_t` is an unsigned integer type.

12.1.5 Return Value

A pointer to the reallocated memory block, which may be either the same as *ptr* or a new location. The type of this pointer is void*, which can be cast to the desired type of data pointer in order to be dereferenceable.

A *null-pointer* indicates that the function failed to allocate storage, and thus the block pointed by *ptr* was not modified.

12.1.6 Example

```
/* realloc example: rememb-o-matic */
#include <cstdio>      /* printf, scanf, puts */
#include <cstdlib>     /* realloc, free, exit, NULL */

int main ()
{
    int input, n;
    int count = 0;
    int * numbers = NULL;
    int * more_numbers = NULL;

    do
    {
        std::printf ("Enter an integer value (0 to end): ");
        std::scanf ("%d", &input);
        count++;

        more_numbers = (int*) std::realloc (numbers, count * sizeof(int));

        if (more_numbers!=NULL)
        {
            numbers=more_numbers;
            numbers[count-1]=input;
        }
        else
        {
            std::free (numbers);
            std::puts ("Error (re)allocating memory");
            exit (1);
        }
    } while (input!=0);

    std::printf ("Numbers entered: ");
    for (n=0;n<count;n++) std::printf ("%d ",numbers[n]);
    std::free (numbers);

    return 0;
}
```

The program prompts the user for numbers until a zero character is entered. Each time a new value is introduced the memory block pointed by numbers is increased by the size of an int.

12.1.7 Data races

Only the storage reference by *ptr* and the returned pointer are modified. No other storage locations are accessed by the call.

If the function releases or reuses a unit of storage that is reused or released by another *allocation or deallocation function*, the functions are synchronized in such a way that the deallocation happens entirely before the next allocation.

12.1.8 Exceptions(C++)

No-throw guarantee: this function never throws exceptions.

Chapter 13

Class

13.1 Initialize const members

There are couple of ways to initialize the const members inside the class.
Definition of const member in general, needs initialization of variable too.

1. First way:

```
class A
{
    static int const a; // declaration
};

int const A::a=10; //defining the static member outside the class.
```

2. Second way:

```
class A
{
    const int b;
    A(int c): b(c) {}// const member initialized in initialization list.
};
```

Chapter 14

Average

14.1 Preliminary average

In the following example, the codes read in a sequence and erase elements having gaps larger than 0.5 mean vaule.

```
#include <vector>
#include <cstdio>
#include <cstdlib>

int main( int argc, char *argv[])
{
    std::printf("------------- Good Morning, My Princess! --------------\n");

    double vals[] = {36.5712, 31.4884, 31.5836, 37.8368, 35.9354, 32.1255, ↩
    160.409, 32.1387, 36.573, 35.3065, 36.561, 37.8515, 142.834, 37.8465, 220.383,↩
     31.5854};
    std::vector<double> seq(vals, vals+sizeof(vals)/sizeof(double));

    double avg;
    int len;

    do
    {
        len=seq.size();
        avg=0.0;
        for( int i=0; i<seq.size(); i++)
            avg += seq[i];

        avg /= seq.size();

        for ( int i=0; i<seq.size(); i++)
        {
            if ( std::fabs( avg-seq[i]) > 0.5 * avg)
                seq.erase(seq.begin()+i);
        }
    }while (len != seq.size());

    std::printf("------------- Good Night, My Princess! --------------\n");
    std::printf("Press Enter to Exit.\n");
    std::getchar();
    std::exit(EXIT_SUCCESS);
}
```

14.2 Group average

```

```

```cpp
#include <cstdio>
#include <cstdlib>
#include <vector>
#include <algorithm>

double GroupMean(int n, double arr[])
{
    double const avg = 0.75;

    // step 1: sort to be ascending.
    std::sort( arr, arr+n);

    // step 2: scan to insert separators with gap larger than 0.75.
    std::vector<int> separator;
    for( int i=0; i<n-1; i++)
    {
        if( (arr[i+1]-arr[i]) > (1.0-avg)*arr[i+1])
            separator.push_back(i+1);
    }

    // step 3: put the remaining  into array.
    separator.push_back(n);

    // step 4: find segments having most elements and its start position.
    int maxCount=separator[0], pos=0;
    for( int i=1; i<separator.size(); i++)
    {
        if( maxCount < separator[i]-separator[i-1] )
        {
            maxCount = separator[i]-separator[i-1];
            pos= separator[i-1]; // in case in the beginning.
        }
    }

    // step 5: find averaged mean.
    double mean=0.0;

    for( int i=pos; i<pos+maxCount; i++)
        mean += arr[i];

    mean /= maxCount;

    return mean;
}

int main( int argc, char *argv[])
{
    std::printf( "Good Morning, My Princess!\n");

    double arr[] = {10.4778,  71.8084, 70.0};

    std::printf("mean = %lf\n", GroupMean( sizeof(arr)/sizeof(double), arr) );

    std::printf("Good Night, My Princess|\n");
    std::getchar();
    std::exit(EXIT_SUCCESS);
}
```

14.3 Xcode

14.3.1 Preprocessor Macros

```
USE_EIGEN USE_GL
```

14.3.2 C++ 11 support

```
Apple LLVM 8.0 - Language - C++
C++ Language Dialect: C++11 [-std=c++11]
```