Module 1

Modern Functional Programming with Visual C++

1 Variables & Constants

1.1 Types & Values

The C++ language provides a limited set of types. It depends on external libraries to provide additional types. A C/C++ compiler comes together with both the Standard C Library and the Standard C++ Library. Operating system libraries are also available to target the specific features of each platform.

C++ Type	Size	Range of Values
[signed] char	1	-128 to +127
unsigned char	1	0 to 255
[signed] short [int]	1/2	-128 to 127 / -32768 to +32767
unsigned short [int]	1/2	0 to 255 / 0 to 65535
[signed] int	2/4	-32768 to +32767 / -2147483648 to +2147483647
unsigned int	2/4	0 to 65535 / 0 to 4294967295
[signed] long [int]	4	-2147483648 to +2147483647
unsigned long [int]	4	0 to 4294967295
[signed] long long [int]	8	–9223372036854775808 to 9223372036854775807
unsigned long long [int]	8	0 to 18446744073709551615
float	4	3.4E +/- 38 (7 digits precision)
double	8	1.7E +/- 308 (15 digits precision)
bool	1	**false** or **true**
<data-type> (pointer)*	2/4/8	0x0000 - 0xFFFF 0x00000000 - 0xFFFFFFFF 0x0000000000000000 - 0xFFFFFFFFFFFFFFFF

The size of certain C++ types can defer depending on the compiler that is being used or the platform that you are building the program for. Standard C++ Library provides a **cstdint** header file containing type definitions where type sizes are fixed. This can be proven by using **sizeof** operator to return the size of the type or variable.

You still need to be wary when using 8-bit types as the base type is still **char** so most functions would still only accept character as input and display a character as output instead of its numerical value. You can still use typecasting to resolve the problem but why add a layer of complexity when you can use 16-bit integer types instead.

Predefined integer types from C++ library: Types1\main.cpp

```
#include <cstddef>  // contains typedef for size_t
#include <cstdint>  // contains typedef for the rest
#include <iostream> // for std::cout and std::endl

void main() {
    int8_t d1 = 127;
    int16_t d2 = 32767;
    int32_t d3 = 2147483647;
    int64_t d4 = 9223372036854775807;
    uint8_t d5 = 255;
    uint16_t d6 = 65535;
    uint32_t d7 = 4294967295;
    uint64_t d8 = 18446744073709551615;
    size_t n1 = sizeof(d1); // 1
    size_t n2 = sizeof(d2); // 2
    size_t n3 = sizeof(d3); // 4
    size_t n4 = sizeof(d4); // 8
    std::cout
        << "sizeof d1 = " << n1 << std::endl
        << "sizeof d2 = " << n2 << std::endl
        << "sizeof d3 = " << n3 << std::endl
        << "sizeof d4 = " << n4 << std::endl;
}
```

In C++/11 you can let C++ detect the type based on the initialization value by using **auto**. When initializing using literal values **int** is the default type for all integer-based values. Use the **u** modifier on the value to get an **unsigned int** instead. If you need specific storage sizes do not use **auto** with integer literal values. Declare the specific fixed-type as shown in the above example instead.

Format	Example	Type	Description
'*<char>*'	'A'	char	an ANSI character
<decimal>[u][l]	123	int	a base-10 integer value
0*<octal>*[u][l]	0123	int	a base-8 integer value
0x*<hexadecimal>*[u][l]	0x3a2b	int	a base-16 integer value
0b*<binary>*[u][l]	0b1011	int	a base-2 integer value
<integer>.*<fraction>*f	1.23f	float	single-precision floating-point value
<integer>.*<fraction>*	1.23	double	double-precision floating-point value
false|true	true	bool	**false** is 0 and **true** is 1
"[*<char>*...]" *(string)*	"Hello!"	char*	zero or more ANSI characters

Using auto in C++/11 for automatic type detection

```
void main() {
    auto v1 = 'A';      // char
    auto v2 = 123;      // int
    auto v3 = 123u;     // unsigned int
    auto v4 = true;     // bool
    auto v5 = 1.9f;     // float
    auto v6 = 1.99;     // double
    auto v7 = "Hello!"; // char *
    auto v8 = v7;       // same as v7 (char *)

    std::cout
        << "v1 is " << typeid(v1).name() << std::endl
        << "v2 is " << typeid(v2).name() << std::endl
        << "v3 is " << typeid(v3).name() << std::endl
        << "v4 is " << typeid(v4).name() << std::endl
        << "v5 is " << typeid(v5).name() << std::endl
        << "v6 is " << typeid(v6).name() << std::endl
        << "v7 is " << typeid(v7).name() << std::endl
        << "v8 is " << typeid(v8).name() << std::endl;
}
```

In C++/11 the array initializer list operator is now the standard initialization operator for all types. This will also clarify the difference between initialization and assignment as different operators are used.

Standard initialization operator in C++/11

```
    auto v1{ 'A' };     // char
    auto v2{ 123 };     // int
    auto v3{ 123u };    // unsigned int
    auto v4{ true };    // bool
    auto v5{ 1.9f };    // float
    auto v6{ 1.99 };    // double
    auto v7{ "Hello!" };// char *
    auto v8{ v7 };      // same as v7 (char *)
```

1.2 Constants & Enumerations

It is recommended to replace fixed values with constants to make code more readable and make it easier for you to replace the values in the future. Even though the correct way to declare a constant is to use the **const** keyword, some developers have chosen to use macros instead. Macros should only be used for conditional compilation of code and should no longer be used for constants. Constants are typed and can be scoped to reduce naming conflicts.

Using macros to replace fixed values: Enums1\main.cpp

```
#include <iostream>

#define FEMALE 0
#define MALE 1

void main() {
    auto g1 = FEMALE;   // auto g1 { Female };
    auto g2 = MALE;     // auto g2 { Male };
    std::cout
        << g1 << std::endl
        << g2 << std::endl;
}
```

Constants can be typed

```
const uint16_t Female = 0;  // const uint16_t Female { 0 };
const uint16_t Male = 1;    // const uint16_t Male { 1 };

void main() {
    auto g1 = Female;
    auto g2 = Male;
            :
}
```

Constants can be scoped

```
struct Gender {
    static const uint16_t Female = 0;
    static const uint16_t Male = 1;
};

void main() {
    auto g1 = Gender::Female;
    auto g2 = Gender::Male;
            :
}
```

If the constants are related you can group them into an enumeration instead. It has auto-numbering and in C++/11 they can also be typed and scoped. Use **enum** to add an enumeration. Use the colon **:** extend operator to specify the base-type. Add **struct** or **class** after the **enum** keyword to scope. You can assign a value manually to each enumerator or let C++ generate the value. The first enumerator will default to 0 and the following enumerator will automatically increment by 1 from the prior one.

A general enumeration of constants

```
enum Gender {
    Female = 0,
    Male = 1
};
```

A type and auto-numbered enumeration

```
enum Gender : uint16_t {
    Female, // implicitly 0
    Male    // implicitly Female + 1 = 1
};
```

A scoped enumeration

```
enum struct Gender : uint16_t {
    Female, // implicitly 0
    Male    // implicitly Female + 1 = 1
};
```

Note that a scoped enumeration actually becomes a type. You can use this to type the variables and arguments that will be used to store or pass values of the enumeration. However there is no automatic value checking and typecasting may be required when using functions that are not build to support the custom type.

An enumeration used as a type

```
void main() {
    Gender g1 = Gender::Female;
    Gender g2 = Gender::Male;
    std::cout
        << (uint16_t)g1 << std::endl
        << (uint16_t)g2 << std::endl;
}
```

1.3 Pointers & References

When you pass or assign a value, pointer, structure or object, a copy is made. This is safe as changing the copy will not affect the original and even if the original no longer exists, you still have a copy.

Copying values through assignment: Pointers1\main.cpp

```
#include <iostream>

void main() {
    int n1 = 10;    // original variable
    int n2 = n1;    // making a copy of the original
    n1 += 10;       // changing the original (does not effect the copy)
    n2 *= 10;       // changing the copy (does not effect the original)
    std::cout
        << "n1 is " << n1 << std::endl
        << "n2 is " << n2 << std::endl;
}
```

Copying structures through assignment

```
struct Time {
    uint16_t hour;
    uint16_t minute;
    uint16_t second;
};

void main() {
    Time t1{ 10, 20, 30 };  // initialize original
    Time t2 = t1;           // make a copy of the original
    t1.hour = 11;           // changing the original
    t2.minute = 22;         // changing the copy
    std::cout
        << "t1 contains "
        << t1.hour << ','
        << t1.minute << ','
        << t1.second << std::endl;
    std::cout
        << "t2 contains "
        << t2.hour << ','
        << t2.minute << ','
        << t2.second << std::endl;
}
```

Be careful when working with arrays. Arrays are passed as pointers so even if a copy is made you are only copying the pointer and not the original array. You are using the pointer to access the original array. Changing the array will affect the data fetched by the pointer and changing the data through the pointer modifies the original array. To make a copy, you have to allocate a new array and copy the elements using **memcpy** function or the safe version **memcpy_s**. Alternatively using the **array** container from the C++ standard library instead of native C++ arrays.

Arrays are passed as pointers

```
void main() {
    int a1[]{ 10, 20, 30, 40 }; // a1 is original array
    auto a2 = a1;               // a2 is pointing to original array
    std::cout << typeid(a1).name() << std::endl;
    std::cout << typeid(a2).name() << std::endl;
    a1[0] = 11; std::cout << a2[0] << std::endl;
    a2[1] = 22; std::cout << a1[1] << std::endl;
}
```

Making a copy of an array

```
int a1[]{ 10, 20, 30, 40 };
int a2[4];
memcpy_s(a2, sizeof(a2), a1, sizeof(a1));
```

Since **array** is a class, you are creating and passing objects. Objects, like structures, are passed as values and not as pointers. So you will be making a copy of the object's internal values. Note that this is a new container type in C++/11.

Using array container objects

```
#include <array>

void main() {
    std::array<int, 4> a1{ 10, 20, 30, 40 }; // a1 is original object
    auto a2 = a1;   // a2 is copy of original object
    std::cout << typeid(a1).name() << std::endl;
    std::cout << typeid(a2).name() << std::endl;
    a1[0] = 11; std::cout << a2[0] << std::endl;
    a2[1] = 22; std::cout << a1[1] << std::endl;
}
```

C/C++ native strings are technically arrays that is terminated with a null character at the end and always passed as **char *****. You can embed special characters in a string by using backslash called as an escape sequence.

String is a null-terminated char array

```
void main() {
    char s1[4];
    s1[0] = 'C';
    s1[1] = '+';
    s1[2] = '+';
    s1[3] = '\0';   // null char to terminate string
    puts(s1);       // a native string function
}
```

Character	Name	Description
\b	backspace	deletes the previous character
\f	form-feed	generates a form-feed on line-printer
\n	newline	generates a line-feed on line-printer
\r	carriage-return	generates a line-feed but on terminal
\t	horizontal-tab	tab to next 8-character aligned column
\"	double-quote	insert a double-quote in a string
\'	single-quote	enter single-quote as a character
\\	backslash	enter backslash as a character / in string
\a	alarm	generates a beep / bell sound
\0	null-char	inserts character with code 0
\0*<octal>*	-	inserts character with octal code
\x*<hexadecimal>*	-	inserts character with hexadecimal code

Strings have the same problem as arrays since they are technically arrays. If you wish to make a copy, you need to allocate a new **char** array and use **strcpy** function or the safer version **strcpy_s**. Alternatively use **string** container class from the C++ library instead of a native C/C++ string. Use **c_str** member function to convert the object to a native C/C++ string if you need to pass it to native string functions. C++ functions from the standard library should accept **string** container objects.

Strings have same problem as arrays

```
void main() {
    char s1[] = "C++";  // s1 is original string
    auto s2 = s1;       // s2 is pointer to original string
    std::cout << typeid(s1).name() << std::endl;
    std::cout << typeid(s2).name() << std::endl;
    std::cout << (s1 == s2) << std::endl; // same memory address
    s1[1] = '-'; puts(s2);  // C-+
    s2[2] = '-'; puts(s1);  // C--
}
```

Making a copy of the string

```
char s1[] = "C++";
char s2[4];
strcpy_s(s2, sizeof(s2), s1);
```

Using string container class

```
#include <string>

void main() {
    std::string s1 = "C++"; // s1 is original object
    auto s2 = s1;   // s2 is copy of original object
    std::cout << typeid(s1).name() << std::endl;
    std::cout << typeid(s2).name() << std::endl;
    std::cout << (s1 == s2) << std::endl;       // same content but
    std::cout << (&s1 == &s2) << std::endl;     // not same memory address
    s1[1] = '-'; puts(s2.c_str());
    s2[2] = '-'; puts(s1.c_str());
    std::cout << s2 << std::endl;
}
```

Alternatively if you not want to make a copy, that obtain a reference instead. This will ensure that no copy is made and you are still accessing the original value or object by using a different variable. However if the original no longer exists, the reference will become invalid. Internally the C++ compiler will pass the memory location just like a pointer but in the C++ language a reference and a pointer is considered opposites. A reference is always regarded as the value rather then the location when used. You can convert between references and pointers in the same manner you obtain the location of a value to assign to a pointer and obtain the value from a pointer.

Referencing original values and objects

```
void main() {
    int n1 = 10;    // n1 stores original value
    int& n2 = n1;   // n2 refers to n1 (no copy is made)
    n1 = 11; std::cout << n2 << std::endl;      // n1 same as n2
    n2 = 22; std::cout << n1 << std::endl;      // n2 same as n1
    std::cout << (&n1 == &n2) << std::endl;     // same memory address
}
```

Conversion between references and pointers

```
void main() {
    int n1 = 10;
    int n2 = 20;
    int& r1 = n1;       // r1 referencing n1
    int *p1 = &n2;      // p1 pointing to n2

    std::cout << &r1 << std::endl;  // location of n1
    std::cout <<  p1 << std::endl;  // location of n2
    std::cout <<  r1 << std::endl;  // value of n1
    std::cout << *p1 << std::endl;  // value of n2

    int& r2 = *p1;  // convert pointer to reference
    int *p2 = &r2;  // convert reference to pointer
}
```

Pointers and references can also be passed as constants. Since no copy is made it will not be safe to pass them to functions if you do not wish the original to be changed. If a function accepts a constant pointer or reference, it is just a confirmation that it will only access the original value but does not change it.

Passing constant pointers and references

```
void p_show(const int *p) {
    std::cout << "value is " << *p << std::endl;
//  *p = 98;    // not allowed to change
}

void r_show(const int& v) {
    std::cout << "value is " << v << std::endl;
//  v = 97;     // not allowed to change
}

void main() {
    int n1 = 99;
    p_show(&n1); std::cout << n1 << std::endl;  // still 99
    r_show( n1); std::cout << n1 << std::endl;  // still 99
}
```

However nothing is truly safe in C/C++. C typecasting and or a C++ casting operator called **const_cast** can be used to remove the **const** modifier when you make a copy of a pointer and reference. For every standard rule there is always a way to break the rule. Programmers like C++ because they can do whatever they like but organizations prefer Java and C# because their programs are less likely to contain bugs. If you wish to write less buggy code in C++, you should not break the rules.

Using casting to remove const modifier

```
void p_show(const int *p) {
    std::cout << "value is " << *p << std::endl;
    //  *p = 98;    // not allowed to change
    int *p1 = const_cast<int*>(p);  // cast to non-constant pointer
    *p1 = 98;   // can change now through a normal pointer
}

void r_show(const int& v) {
    std::cout << "value is " << v << std::endl;
    //  v = 97;     // not allowed to change
    int &v1 = const_cast<int&>(v);  // cast to non-constant reference
    v1 = 97;    // can change now through a normal reference
}

void main() {
    int n1 = 99;
    p_show(&n1); std::cout << n1 << std::endl;  // no longer 99
    r_show( n1); std::cout << n1 << std::endl;  // no longer 99
}
```

Literal native strings are compiled into read-only memory. You need to make sure that any pointer to a literal native string must be constant. Trying to use a normal pointer to change the literal string will generate a run-time error. Alternatively make a copy of the literal string if you need to modify its content.

Trying to modify read-only string

```
void main() {
    char *s1 = "Hello!";    // pointer to read-only memory
    puts(s1);               // works when only accessing the string
    s1[0] = 'M';            // fails when trying to change it
}
```

Correct way of using literal strings

```
const char *s1 = "Hello!";  // pointer to read-only string (cannot change)
char s2[] = "Hello!";       // making a copy of read-only string (can change)
//  s1[0] = 'M';            // compiler disallow changes
s2[0] = 'M';                // compiler allow changes
puts(s1);                   // not changed
puts(s2);                   // changed
```

1.4 Dynamic Memory

There are three storage areas where memory can be allocated to store values, objects and pointers; **static**, **stack** and **dynamic**. Static memory is safe since the memory is allocated at the start of the program and exists until the program ends. Any pointer or reference to static memory is always valid. Global variables used static memory by default. Static memory is automatically initialized to 0 before the program runs so you know exactly what is the default value. Global variables can be accessed directly from any function by name.

Using global variables: Dynamic1\main.cpp

```
#include <iostream>

int g1;         // global variable (static memory, defaults to 0)
int g2 = 123;   // global variable (static memory, initialized to 123)

void main() {
    std::cout
        << g1 << std::endl
        << g2 << std::endl;
}
```

Stack memory is only allocated within the function for arguments, local variables and inline objects. Even though stack memory is initialized at the beginning of a program, it is not initialized to 0 and will not be reinitialized when the memory is recycled at the end of the function. Anything allocated in stack memory can only remain until the end of the function. Any references or pointers to stack memory is still valid as long as the function has not completed yet. Functions must never return a pointer or reference to stack memory as they become automatically invalid when the function returns. Local variables use stack memory by default but you can use the **static** keyword to allocate static memory instead.

Using local variables

```
void test_locals() {
    int l1 = 0;     // using stack memory (no default, must initialize)
    static int l2;  // using static memory (has default value)
    std::cout << ++l1 << std::endl;  // does not retain value
    std::cout << ++l2 << std::endl;  // retains value
}

void main() {
    test_locals();
    test_locals();
    test_locals();
}
```

Regardless of what memory you are using for local variables, they can only be directly accessed in the function that are declared in. However the function can pass the local variable by value, by reference or by pointer to other functions that it calls.

Passing local variables to other functions

```
void updatelocal(int& v) {
    std::cout << ++v << std::endl;
}

void test_locals() {
    int l1 = 0;     // using stack memory (no default, must initialize)
    static int l2;  // using static memory (has default value)
//  std::cout << ++l1 << std::endl;  // does not retain value
//  std::cout << ++l2 << std::endl;  // retains value
    updatelocal(l1);
    updatelocal(l2);
}
```

You can create temporary objects that only last for one statement. It is created during the statement and will only exist until the end of the statement. They are called inline objects. Use the class to create and initialize an object without assigning to a variable to create an inline object.

Creating inline string objects

```
#include <string>

void show_string(std::string& message) {
    std::cout << message << std::endl;
}

void main() {
    std::string s1 { "Hi!" };   // not an inline object
    show_string(s1);    // Hi! string not destroyed at end of statement
    show_string(std::string{ "Hello!" });       // Hello! string destroyed
    show_string(std::string{ "Goodbye!" });     // Goodbye! string destroyed
    show_string(std::string{ "Sayonara!" });    // Sayonara! string destroyed
}   // Hi! string destroyed at end of function
```

The C++ compiler will generate code to allocate and release memory at the right time for static and stack memory. We do not need to worry about memory leaks as long as the data is allocated in these areas of memory. If you wish to control when memory is allocated and released you can use dynamic memory. Dynamic memory is allocated by using the **new** operator and a pointer to that memory is returned. You must keep the pointer or else you will not be able to release the memory later. The C++ compiler does not generate code to release dynamic memory until the end of the program. You must make sure that you use the **delete** operator to release dynamic memory you no longer wish to use or else memory leaks would occur. However all memory would still be released back to the operating system at the end of the program.

You are free to choose whether to store pointers in static or stack memory as it does not matter which function allocates the memory and which function releases it as long as all the functions have access to the pointer. All you need to do is to make sure that the pointer is retained from the time you allocated the dynamic memory to the time it is released.

Allocating and releasing dynamic memory

```
int *array1;    // global pointer to dynamic array

void create_array(int size) { array1 = new int[size]; }
void delete_array() { delete array1; }

void main() {
    // dynamic array has not been allocated yet, do not access it here
    create_array(2); // use another function to create array
    array1[0] = 10; // access array through global pointer
    array1[1] = 20;
    std::cout << array1[0] << std::endl;
    std::cout << array1[1] << std::endl;
    delete_array(); // use another function to destroy the array
    // dynamic array has already been released, do not access it here
}
```

You should take extra precautions when working with dynamic memory to reduce the chance of bugs. No dynamic memory can be allocated at address 0 so you can use it as an indication of an invalid pointer. Pointers should always be initialized to 0 so that functions will know whether memory has been allocated or not. The function can then choose not to reallocate the memory or release the previously allocated memory. The following shows better written functions that works with valid and invalid pointers. A pointer is not automatically invalidated when you use the **delete** operator. It will still contain the memory address even though that memory has already been released. It is crucial for you to manually invalidate the pointer for your functions to work properly with dynamic memory.

A better dynamic memory allocation function

```
void create_array(int size) {
    if (array1 != 0) delete array1; // release previously allocated memory
    array1 = new int[size]; // allocate new dynamic memory
}
```

A better dynamic memory release function

```
void delete_array() {
    if (array1 != 0) {  // check to make sure memory has been allocated
        delete array1;  // release the allocated memory
        array1 = 0;     // invalidate the pointer
    }
}
```

A pointer containing address 0 is called a null pointer. Instead of using 0 in your code, you can use a **NULL** macro that has already been defined in the Standard C Library. However you would need to include at least one header file from standard libraries to ensure that it has been defined. Alternatively you can define manually if the macro is not already defined.

Manual definition of NULL macro

```
#ifndef NULL
#define NULL 0
#endif
```

However in C++/11 this is no longer necessary as there is now a **nullptr** keyword for assigning and checking null pointers. There is no more need for the NULL macro or to define your own. You can now use the keyword to assign and test null pointers. Also remember that in C/C++, whatever is zero is considered as **false** and not zero will be considered as true so a null pointer is false and a valid pointer would be true. This will simplify your checking code without even using **nullptr**.

Using boolean logic and nullptr

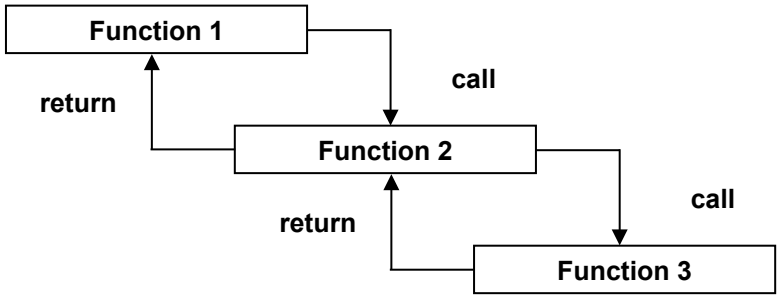
```
void create_array(int size) {
    if (array1) delete array1;  // release previous dynamic memory
    array1 = new int[size];     // allocate new dynamic memory
}

void delete_array() {
    if (array1) {       // checking for valid pointer
        delete array1;  // release dynamic memory
        array1 = nullptr;   // invalidate pointer
    }
}
```

2 Functions & Arguments

2.1 Functions

A function contain code that is executed when the function is called. One function can call other functions. The function ends when all the statements have been executed or a **return** statement is encountered. Control will then be transferred back to the caller function.

Calling functions from other functions



A function is called by name. All symbolic names in C++ can contain alphabets, digits and underscore but cannot begin with a digit. A function can choose to return a value by declaring the data type of the return value in front the function name. Use **void** for a function that has no return value. A function can receive input data from the caller by declaring one or more arguments inside the required braces after the name.

Function implementation

```
data_type function_name([argument, ...])
{
    [statement; ...]
}
```

Argument declaration

```
data_type argument_name
```

Example function implementation: Functions1\main.cpp

```
double payment(double loan, double rate, int term) {
    return (loan + loan * rate * term) / (term * 12);
}
```

The most important function in a C++ program is **main**. This function is automatically called when a program is executed. This is considered as the entry point function of a program. You cannot build an executable program without a main function.

Simplest main function

```
void main() {
    // statements to be executed when the program starts
}
```

To reuse the same functions across multiple programs they can be build into libraries. C++ compilers come together with a Standard C Library and a Standard C++ Library containing hundreds of functions to use in your programs immediately. C++ compiler requires a function to be either implemented or declared before use. Declarations are normally placed into a header file. You must include the header files for the functions that you need to use before calling them.

Calling external functions

```
#include <stdio.h>  // contains declaration for puts function

void main() {
    puts("Hello C++!"); // call puts function from Standard C Library
}
```

2.2 Passing Arguments

An argument can be passed to a function by value reference or by pointer, depending on the declaration of the argument. When it is passed by value, a copy of the value in the variable is made on the function stack. When a function accesses the value, it is accessing the copy and not the original variable and changing it does not affect the original variable the value comes from. When function ends, the copy is destroyed since the function call stack memory is recycled at the end of the function.

Passing arguments: Arguments1\main.cpp

```
void foo1(int fooVar) { // passing by value
    fooVar = 99;        // changing copy of value
}

void foo2(int *pFooVar) {   // passing by pointer
    *pFooVar = 98;  // accessing the original value
}

void foo3(int& fooVar) {    // passing by reference
    fooVar = 97;    // changing the original value
}
```

You may also explicitly pass address of the variable by using the pointer (*****) operator. However, you must then use the same operator within the function code to access the value from the address. Since you are passing the address, recycling the function call stack memory will only destroy the address. To pass by reference you can then use the reference (**&**) operator. The actual address of the variable will be implicitly passed over and no copy of the value will be made. Access to the value is done by using the address of the original variable. Basically this is just a way that we can implicitly pass the address of a variable but does not have to use the pointer syntax.

Passing arguments to functions

```
void main() {
    int var1 = 100;
    foo1(var1); cout << var1 << endl;   // 100
    foo2(&var1);cout << var1 << endl;   // 98
    foo3(var1); cout << var1 << endl;   // 97
}
```

Passing by pointer or reference is quite important for large structures and objects as no copy will be made. If you pass by value, a copy of the structure or object has to be made first before the function can run. It would take a long time and the copy may not work or can cause problems for a complex object. However, passing by pointer or reference would allow the structure or object to be changed. If we want to pass structures and objects only for input, we can always declare the reference or pointer as constant.

Passing constant pointer & constant reference

```
void foo5(const int *pFooVar) { // passing by pointer for input only
    *pFooVar = 96;              // this cannot compile!
}

void foo6(const int& fooVar) {  // passing by reference for input only
    fooVar = 95;                // this cannot compile!
}
```

2.3 Function Pointers

It is possible to store the address of a function in a variable and then use the variable to call the function. We normally call this as a virtual function. We can use **typedef** to define the function pointer type. If you are using a C++/11 or later compiler you no longer will need to define pointer type and use **auto** instead. The **auto** keyword lets the C++ compiler detect the variable type automatically.

Declaring function pointer type Virtual1\main.cpp

```
typedef int (*myfuncptr)(int, int);
```

Different functions but following the same signature

```
int add(int x, int y) { return x + y; }
int sub(int x, int y) { return x - y; }
int mul(int x, int y) { return x * y; }
```

C++/11 can recognize the different between a normal pointer and a function pointer so if you attempt to use the pointer as a function, it will automatically dereference the pointer for you. The example shows manual and automatic dereferencing.

Assigning function pointers and calling functions through pointer

```
void main() {
    myfuncptr p;
//  p = add; std::cout << (*p)(10,20) << std::endl;
//  p = sub; std::cout << (*p)(10,20) << std::endl;
//  p = mul; std::cout << (*p)(10,20) << std::endl;
    p = add; std::cout << p(10,20) << std::endl;
    p = sub; std::cout << p(10,20) << std::endl;
    p = mul; std::cout << p(10,20) << std::endl;
}
```

Since a pointer is just a value, you can pass functions to other functions as arguments using function pointers. Functions can then be called through the pointers as shown below.

Accepting function pointer as argument

```
void display(myfuncptr f) { std::cout << f(10, 20) << std::endl; }

void main() {
    display(add);
    display(sub);
    display(mul);
}
```

No pre-declaration of pointer type

```
void display(int f(int, int)) { std::cout << f(10, 20) << std::endl; }

void main() {
    auto f = add;
    std::cout << f(10,20) << std::endl; f = sub;
    std::cout << f(10,20) << std::endl; f = mul;
    std::cout << f(10,20) << std::endl;
    display(add);
    display(sub);
    display(mul);
}
```

A C++ **using** keyword while commonly used to bring members out of a named scope into the global scope, can also be used to provide an alias to type names. You can use this feature to provide an alias to pointer types as well. This is only an alias so it does not physically generates a new type.

Importing out certain members from a named scope a.k.a namespace

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

void display(int f(int, int)) {
    cout << f(10, 20) << endl;
}
```

Importing all members of a namespace

```
using namespace std;
```

Assigning an alias to a pointer type

```
using myfuncptr = int(*)(int, int);

void display(myfuncptr f) {
    cout << f(10, 20) << endl;
}
```

2.4 Lambda Expressions

Since functions can be called through pointers, in C++/11 you can now directly assign code to variables in the format shown below, called as lambda expressions. These are functions that can be called through the variables that you assign to. Return type can be auto-detected and left out. Capture list is used to capture additional local variables so they are available to the lambda expression during execution. Use = to capture all local variables by value or & to capture by reference.

Lambda expression syntax

```
[capture_list](arguments) -> return_value_type { statements; }
```

Using lambda expressions: Lambda1\main.cpp

```
auto add = [](int x, int y) -> int { return x + y; };
auto sub = [](int x, int y) { return x - y; };
auto mul = [](int x, int y) { return x * y; };
```

Directly passing functions as arguments

```
display([](int x, int y) { return x + y; });
display([](int x, int y) { return x - y; });
display([](int x, int y) { return x * y; });
```

Capturing local variables

```
int v = 0;
auto add = [&v](int x, int y) { return v += x + y; };
auto sub = [&v](int x, int y) { return v += x - y; };
auto mul = [&v](int x, int y) { return v += x * y; };
cout << add(10, 20) << endl;// 30
cout << sub(10, 20) << endl;// -10 = 20
cout << mul(10, 20) << endl;// +200 = 220
cout << v << endl;
```

2.5 Function Overloading

You can have functions with the same name as long as the number of arguments are different or the type of arguments are different. This reduces the number of function names that developers have to remember. Of course you should only overload those that have similar functionality but has to provide support for multiple types.

Overloading functions: Overloading1\main.cpp

```
// swap function for int
void swap_vars(int& v1, int& v2) {
    int vt = v1; v1 = v2; v2 = vt;
}
// swap function for double
void swap_vars(double& v1, double& v2) {
    double vt = v1; v1 = v2; v2 = vt;
}
// swap function for string
void swap_vars(const char *& v1, const char *& v2) {
    const char * vt = v1; v1 = v2; v2 = vt;
}

#include <iostream>
using namespace std;

void main() {
    int n1 = 66, n2 = 99;
    double d1 = 1.99, d2 = 99.1;
    const char *s1 = "Hello!", *s2 = "Goodbye!";
    swap_vars(n1, n2);  // swap_vars(int&,int&)
    swap_vars(d1, d2);  // swap_vars(double&,double&)
    swap_vars(s1, s2);  // swap_vars(const char*&,const char*&)
    cout
        << n1 << ',' << n2 << endl
        << d1 << ',' << d2 << endl
        << s1 << ',' << s2 << endl;
}
```

2.6 Generic Functions

If the code is the same but only the type is different, then you use generic functions that can be overloaded automatically for you by the compiler. In C++ you can create templates to overload generic functions. For example you may have wrote a generic function that returns the maximum between two **int** arguments.

Example int function

```
int max_value(int a, in b) { return a > b ? a : b; }
```

The above function only works with **int** type arguments. You would have to overload the above function to support another data-type. However, it will be impossible for us to overload the function to support every data-type unless we implemented a generic function instead. A generic function is defined once, and will automatically be used to generate code to support every data-types. You can use C++ templates to generate a generic function. Following example uses template to define a **max** function returning the maximum of two arguments independent of the data type. Generic functions have to be distributed in source code form usually in header files.

Defining a generic function: my_templates.h

```
#pragma once
template <typename X>
X max_value(X a,X b) { return a > b ? a : b; }
```

You can define generic arguments in the template code as data-types by using the **class** or **typename** attribute. In the above example **X** is used to indicate a data type, and we can use it anywhere in the code. At this point of time, there is no indication on what **X** is but some kind of data-type. Thus no code is generated until the template is instantiated. For a generic function, we can instantiate the template by just using the generic function directly in our code. If C++ can detect the type from arguments then there is no need to specify the exact type when calling the function.

Instantiating generic functions

```
char v1 = max_value('A','B');    // max_value<char>('A', 'B');
int v2 = max_value(10, 20);      // max_value<int>(10, 20);
double v3 = max_value(40.20, 18.00);// max_value(double,double)(40.20,18.00);
```

What about operations that would require using more than one type? You can indicate as many generic type arguments as you like in a template. Thus you may have more than one type but they can also be the same type. The types can be used anywhere in the function.

Multiple generic types

```
template <typename X,typename Y>
X add(X arg1,Y arg2) { return (X)(arg1 + arg2); }
```

Instantiating with multiple types

```
char a = add('A', 2);        // add<char,int>('A', 2);
long b = add(10, 69.99);     // add<int,double>(10, 69.99);
```

You can now easily swap any kinds of variables by defining the following template to instantiate a generic swap function. You will not need to write another swap function again. Just include the header file that contains the template and call the function in your code.

Defining a generic swap function: my_templates.h

```
template<typename T>
void swap_vars(T& v1, T& v2) {
    T vt = v1; v1 = v2; v2 = vt;
}
```

3 Unicode Programming

3.1 Supporting ANSI & Unicode

The Windows operating system has Unicode support since Windows NT. ANSI is still supported for backward compatibility for old Windows applications that was built for Windows 95, 98 and ME. Each ANSI character takes up one byte while each Unicode character takes up two bytes. In C++, you can use **char** type for an ANSI character and **wchar_t** for Unicode. You can also use the **CHAR** and **WCHAR** macro defined for these types. Use **TCHAR** if you wish to be able to compile a program to either an ANSI or Unicode. If the **UNICODE** symbol is defined before compilation, then it will be translated to **wchar_t** otherwise it will be translated to **char**. Include **TCHAR** header if you use this type. Include **Windows** and **tchar** header files for these types.

ANSI & Unicode character buffers: Unicode1\main.cpp

```
char b1[128];       // 128 bytes ANSI char array (128 bytes)
wchar_t b2[128];    // 128 bytes Unicode char array (256 bytes)
TCHAR b3[128];      // either ANSI or Unicode char array
```

By default C++ literal strings are always compiled to ANSI. If you wish to use a Unicode literal string, you have to attach an **L** prefix. To be able to compile the string to either ANSI or Unicode, you can use the **TEXT** or **_T** macro which will attach an **L** prefix to the string only if the **UNICODE** symbol has already been defined before compilation. For pointers to strings and character arrays, use **LPSTR** and **LPCSTR** for ANSI, **LPWSTR** and **LPCWSTR** for Unicode and lastly **LPTSTR** and **LPCTSTR** for neutral.

ANSI & Unicode literal strings

```
LPCSTR  p1 = "This is an ANSI string.";
LPCWSTR p2 = L"This is an Unicode string.";
LPCTSTR p3 = TEXT("This can be an ANSI or Unicode string.");
LPSTR s1 = b1; LPWSTR s2 = b2; LPTSTR s3 = b3;
```

The C++ library provides a set of string functions for manipulating ANSI strings. Microsoft has added additional functions to support Unicode strings. The name of the functions has additional **w** character for wide-character support.

ANSI & Unicode string functions

```
strcpy(s1,p1);      // ANSI string function
wcscpy(s2,p2);      // Unicode string function
_tcscpy(s3,p3);     // either ANSI or Unicode function
```

Visual C++ also provide a set of non-standard safer set of functions that forces you to pass in the length of the buffer to ensure that there is no buffer overrun. An exception will be thrown immediately if there is an overrun.

Safe version of string functions

```
strcpy_s(s1,sizeof(b1),p1);
wcscpy_s(s2,sizeof(b2) / 2,p2);
_tcscpy_s(s3,sizeof(b3) / sizeof(TCHAR), p3);
```

The Windows API will also provide two functions for operations that accept any strings or character array, one that is postfix with **W** and another with **A**. When the **A** function is called, ANSI strings passed to the function will be converted into Unicode and will be then forwarded to the **W** function which performs the actual task. When any string is passed back by the **W** function will be converted back to ANSI before returning back from the **A** function. Thus calling functions that end with **W** will be faster. You can leave out the postfix altogether and let the compiler determine which function to call based on if the **UNICODE** symbol has been defined. You can use the Dependency Walker utility (depends.exe) to examine the Windows API Libraries to see the function names.

Calling ANSI function from kernel32.dll

```
void GetComputerName1() {
      char computerName[256];
      DWORD dwLen = sizeof(computerName);
      GetComputerNameA(computerName, &dwLen);
      MessageBoxA(NULL, computerName, "Computer Name", MB_OK); }
```

Calling Unicode function

```
void GetComputerName2() {
      wchar_t computerName[256];
      DWORD dwLen = sizeof(computerName) / 2;
      GetComputerNameW(computerName, &dwLen);
      MessageBoxW(NULL, computerName, L"Computer Name", MB_OK); }
```

Calling either ANSI or Unicode

```
void GetComputerName3() {
      TCHAR computerName[256];
      DWORD dwLen = sizeof(computerName) / sizeof(TCHAR);
      GetComputerName(computerName, &dwLen);
      MessageBox(NULL, computerName, TEXT("Computer Name"), MB_OK); }
```

You should use **TCHAR**, **LPTSTR**, **LPCSTR** types and **TEXT** macro so that you can compile a Win32 application to either ANSI or Unicode. You can select the **Character Set** to use in **General** section of the project **Configuration** before compilation. In the latest version of Visual C++, it defaults to Unicode.