Introduction

The creator of C++, Bjarne Stroustrup, states: “C++ is a general-purpose programming language ... that is a better C, supports data abstraction, supports object-oriented programming and supports generic programming.” [Stroustrup 2000] It was because of these qualities, in particular object-oriented programming support, that in 1994 Symbian (then part of Psion) adopted C++ when rewriting their 16-bit operating system SIBO to create EPOC32, which has since evolved into Symbian OS.

Symbian OS C++ is built on top of C++ with some deliberate omissions, including exceptions, templates and the standard template library. It also extends or, more accurately, complements C++ by providing strong coding conventions, small-memory management techniques and highly optimized algorithms for mobile phones. To program in C++ on Symbian OS, developers require strong skills in mainstream C++.

This chapter covers the required language basics for the ASD exam: types, statements, functions and a little about the tool chain. It touches on some object-oriented (OO) syntax but only where needed, as this is dealt with in more detail in Chapters 2 and 3.

1.1 Types

This section covers fundamental types, including pointers, references and arrays. There are some very basic concepts covered here along with some syntactical foibles. Candidates are not expected to memorize the intricacies of the syntax but should not be confused by any unusual syntax.
Critical Information

Definition

Doing anything of practical use in C++ involves variables and functions. A variable requires a name, a type and, as a rule, a collection of meaningful actions that may be carried out on it.

Informally, types specify what set of values a name can contain, along with what operations are valid for those values. For example, an integer type allows only whole number values and has plus and subtraction operations (among others) associated with it.

All values in C++ require memory. The language itself does not specify how much memory – that is an implementation detail – but typically an int is represented by a single 32-bit word (4 bytes). This is about as basic as it gets:

```c++
int n = 42;
```

The identifier `n` is the name of a variable of type `int` and is initialized with the value of 42 (typically held in 4 bytes of storage).

Basic types

The following basic types are built into the C++ language and are known as fundamental types:

- `bool` – Boolean
- `char` – character
- `int` – integer
- `double` – floating-point number.

There is also `wchar_t`, the wide character type, not covered here.

The first three types, `bool`, `char` and `int`, are known as integral types, values that do not have a fractional part. Integral types may be promoted or cast to larger types without any loss of precision, for example from a `char` to an `int`. For assignments between integral and floating-point types (`double`), precision cannot be guaranteed; for example, the fractional part of a `double` is rounded on assignment to an `int`.

Integral types and floating-point types are called arithmetic types, types that allow mathematical manipulations. Arithmetic types can be mixed freely in assignments and expressions, but this practice is not recommended due to undefined behavior and potential loss of precision.

Integral types may be augmented by the keywords `signed`, `unsigned`, `short` and `long` to allow better granularity of the storage size of the type. The `unsigned` keyword, for example, frees up the
complement bit, allowing larger positive numbers to be represented in the same amount of memory; unsigned short char typically represents an 8-bit ASCII character.

**Typedef**

To provide developers with a degree of flexibility both in the practicalities of naming and as a porting aid, C++ provides the `typedef` keyword to allow a type to have an alias.

```cpp
typedef unsigned long int TUint32
```

TUint32 is less of a mouthful than unsigned long int and, when porting to another system, if the underlying implementation of long is less than 32 bits, only the typedef declaration need be modified to accommodate a larger type.

It should be noted that typedefs are not distinct types in themselves but synonyms or aliases. There are no specific operations or type checks for TUint32, only for the existing type it substitutes.

**Constants and enumerations**

The keyword `const` provides a mechanism to reduce errors caused by inadvertently reassigning a value to an identifier. A const identifier has to be initialized when it is declared and cannot be reassigned any other value.

```cpp
const int n = 42; // n has to be initialized and cannot be reassigned
```

C++ also supports a legacy method for providing named constants, the `#define` preprocessor directive (see the information about preprocessors in Section 1.6).

```cpp
#define N 42
```

This is a brute force approach to constants; the preprocessor is executed before the C++ compiler and replaces all the occurrences of the identifier N with the value 42. The problem with this is twofold: there is no scope control and no type-checking. Using the const definition, n can only be defined once, is clearly an int, and has a well-defined scope. With the preprocessor directive, there is no concept of scope and N may be redefined at any time to contain another value of any type.

The final type of constant that C++ supports is a user-defined type enumeration:
### enum TPriority

```cpp
enum TPriority {
    EPriorityIdle = -100,
    EPriorityLow = -20,
    EPriorityStandard = 0,
    EPriorityUserInput = 10,
    EPriorityHigh = 20
};
```

`TPriority priority = EPriorityLow;`

`priority = EPriorityHigh; // (a) Fine: priority itself is not a const`

`priority = -10; // (b) Error: -10 is not of type TPriority`

`priority = TPriority(-10); // (c) Fine: -10 is within the enumeration`

`// range and may be explicitly converted`

`priority = TPriority('a'); // (d) Fine! 'a' is a char - an integral`

`// type`

`int i = EPriorityIdle; // (e) Fine`

In the above example, TPriority is the name of a type that has a set of five constant enumerators with assigned values. This is not mandatory; if no values are assigned, the first enumerator value is 0 and each subsequent enumerator increments by 1.

The enum variable `priority` can only be assigned constant values of the TPriority type, although integral types can also be assigned TPriority values. See the comments in the example.

See Chapter 10 for information on the Symbian OS treatment of `const` values in DLLs.

### Pointers

C++ provides a very simple derived type, the pointer type, which has led to some of the most powerful and, arguably, difficult to understand code ever written. For every fundamental or user-defined type there is an associated type that points to it; in pseudo-formal language, for every type `T` there is the pointer type `T*`. A variable of type `T*` simply holds the address of a variable of type `T`.

```cpp
short int n = 42;
int* p = &n;  // The address of short int variable n
short int m = *p; // p is dereferenced and m is assigned n, i.e. 42
```

The variable `n` has the short integer value of 42. The address of `n` is retrieved by the address-of operator, `&`, and is assigned to the pointer variable `p`. To retrieve the value of `n` from `p`, it is a matter of dereferencing the pointer using the dereference operator, `*`.

Note the use of the `*` operator: in a declaration statement or function parameter list, the `*` symbol signifies a pointer type whereas, in expressions, it is the dereference operator.
It is possible to have pointers to pointers or multi-level pointers.

```cpp
// carrying on from the above example
int** pp = &p; // pp contains the address of p which contains the
               // address of n which contains the value 42
m = **pp; // deref pp, deref p and assign the value of n to m
```

Thus `pp` contains the address of the `p` pointer, which in turn contains the address of the `n` identifier containing the value 42. Multi-layering can in theory continue indefinitely.

**Null or zero**

It is possible to declare an uninitialized pointer, or for a pointer to unintentionally point at some random part of memory. There is no way to determine at run-time whether a pointer contains an illegal address. To reduce this possibility, C++ has made the number zero (0) special for pointers.

A pointer containing the value 0 is not pointing at an object but is a **null pointer**. It is usual practice to declare a global constant to clarify the null value:

```cpp
const int NULL = 0;
```

**Pointer arithmetic**

Pointer arithmetic is relative to the size of the type that is **pointed to**.

```cpp
short int n; // size of short int is 2 bytes
short int* p = &n // p contains address of n
p = p+1; // p has incremented by 2 bytes

int m; // size of int is 4 bytes
short int* q = &m; // q contains address of m
q = q+1; // q has incremented by 4 bytes
```

So, for `short int* p`, `p=p+1` increments the pointer by 2 bytes, whereas for `int* q`, the increment is 4 bytes.

This behavior is applicable to all pointer arithmetic operations including the `++` and `--` operators (see Section 1.3).

**Arrays**

A C++ array is a contiguous piece of memory containing individually addressable values which may be accessed using the offset operator, `[]`. In expressions, the `[]` operator takes an index to access the value at that position in the array. C++ arrays are zero-indexed.
An array behaves in the same *type-size-safe* manner as a pointer, so an array of three *short int* values requires 3*2 bytes whereas an array of three 4-byte *int* values requires 3*4 bytes. Pointers and arrays are interchangeable in some respects:

```cpp
int* ptr = array; // ptr contains the address of the 1st
// element
res=*(ptr+1)+*(ptr+2); // add all the elements - it can soon get
// confusing
```

The statement `ptr = array;` is semantically the same as `ptr = &array[0];`. This is because C++ does not support operations on entire arrays, only on individual array elements; any mention of an array name in an expression is interpreted by the compiler as a pointer to the address of its first element.

```cpp
int aArray1[3];
int aArray2[3];
aArray1 = aArray2; // Illegal! No array-level operations supported
```

**Multidimensional arrays**

In the same way as there are pointers to pointers, there can be arrays of arrays or, more accurately, multidimensional arrays.

```cpp
int matrix[3][3]={{1,2,3},{4,5,6},{7,8,9}};
int n = matrix[1][1]; // n == 5
```

The `matrix` variable is a 3 by 3 array, that is three arrays each containing three integer values. Note that in the `matrix` declaration only the first dimension may be undefined; the second and any subsequent dimension sizes must be known.

See Section 1.4 for notes and comments about argument passing to functions.

**References**

A reference is an alias or alternative name for an existing object. The ampersand symbol (`&`) indicates a reference type in a declaration. The following example shows the behavior and use of references:

```cpp
int n = 42; // n contains 42
int& ref = n; // ref is an alias for n
```
References are different from pointers in a number of ways:

- References must be initialized (when declared they must refer to an existing object) whereas pointers may be declared uninitialized.
- Once initialized, a reference cannot be changed, but a pointer may be reassigned.
- There are no direct operators that act on references, only on the value referenced. A pointer has operators that act directly on it (arithmetic and dereference).
- There is no NULL value for a reference.

**Const pointers**

Const pointers play an intricate part in C++ programming, often requiring a keen eye to ensure that the intended semantics and behavior are correct. Specifically, a const pointer is not a pointer to const.

```c++
const char* p1 = "Mouse"; // pointer to const identifier
cchar* const p2 = "Small"; // const pointer to non-const variable
const char* const p3 = "Horse"; // const pointer to a const identifier
p1 = "Elephant"; // Legal to reassign non-const ptr
p2 = "Huge"; // Illegal reassignment of const ptr
```

In the above example, `p1` is a pointer that points to the constant value "Mouse", but it may be reassigned to point to another const char value. `p2` is a constant pointer to a non-constant value and cannot be reassigned (similar to a reference). `p3` is itself constant and also points to a constant value.

```c++
char* p4 = "Cat"; // pointer to non-const declaration
p1 = p4; // pointer to const assigned a non-const variable
p4 = p1; // Error: illegal attempt to downgrade a const
```

C++ allows a pointer to const to point to a non-const value. This initially appears counterintuitive, but what it means is the value being pointed to will be treated as a const. However, a pointer to a non-const cannot be assigned the value of a pointer to const, as that would remove the constness of the pointer to const type.

It is worth noting the following syntactical equivalences:

```c++
char const* p5 = "Duck"; // pointer to const, the same as p1
char const* const p6 = "Goat"; // const pointer to const char, the same as p3
```
Exceptions and Notes

The `mutable` keyword, which can override `const`, is not covered here.

Exam Essentials

- Understand that C++ has a number of different types in various categories for example integral, arithmetic
- Recognize `typedef` as defining a synonym for an existing type but not a new type in itself
- Recognize enumeration types as user-defined value sets
- Specify the advantages of `const` over `#define`
- Understand the use and properties of the C++ reference type
- Specify the difference between pointers and references
- Understand the semantics of pointer arithmetic
- Recognize pointer operations and the purpose of the `NULL` pointer value
- Differentiate `const` pointers and pointers to `const`

References

[Dewhurst 2005 Items 5, 7 and 8]
[Stroustrup 2000 Chapters 2 and 4, Sections 5.1–5.6 and Sections C.1–C.7]

1.2 Statements

Critical Information

*Declarations and definitions*

This section broadly covers the C++ rules for declaration and definition. It does not go into too much specific detail as this is dealt with in Chapter 2.

Before an identifier or a function can be used, it has to be declared. A *declaration* does not have any execution expressions or storage requirements, and an entity may be declared any number of times throughout a program as long as the declarations are consistent. An entity *definition* contains the expressions and storage requirements for the identifier and an entity may only be defined once so that it does not have more than one possible *meaning*. 
At its most basic, declaring a variable is a matter of associating an identifier with a type, for example `int n` (see Section 1.1). In this case, the declaration is also the definition as `n` requires storage (4 bytes).

Function declarations and definitions are typically separate, allowing implementation hiding and separate compilation (see Section 1.6):

```c
int Add(int a, int b); // Function declaration
// ...
int Add (int a, int b) // Function definition
{
    return a+b;
}
```

The `extern` specifier, when added to the start of a declaration, tells the compiler not to allocate memory for the identifier as it is defined elsewhere:

```c
extern int myInt; // Declaration, no memory is allocated to myInt
//...
int myInt = 42;   // myInt definition
```

Scope

Once declared, an identifier has a scope or lifetime. Scope is the span of code in which an identifier is usable. In C++, scope has several levels of granularity: file or global scope, function scope, block scope, class scope and namespace scope. Class, namespace and function scope are dealt with in Chapter 2.

In file or global scope, the identifier is visible from its declaration to the end of the file. The `extern` keyword enables the same identifier to be declared visible in more than one file.

Block scope is the most common unit of scope. A block consists of zero or more statements enclosed by braces (or “curly brackets”), `{}`. Blocks may be nested. The following example shows two nested `if` blocks:

```c
int m = 32;
if (m>0)
    { // Outer block scope
        const int n = 42;
        if (n>m)
            { // Inner block scope
                int i = n;  // outer n is in scope, i is 42
                ...
                const int n = 52; // Legal
                i = n;           // inner n is in scope, i is 52
            }
        // i and inner n are out of scope, outer n is in scope
    } // outer n is out of scope, m is still visible
```

It is important to note that the outer block identifiers are scoped within the inner block, but a declaration (and definition) of the same name in
the inner block overrides the outer scope identifier until the inner block closes and the inner identifier goes out of scope.

**Conditional statements**

C++ supports two conditional or selection statements, if and switch, in which a condition is tested for and the result determines which action (if any) is carried out.

```cpp
if (condition) statementOne
```

StatementOne is only executed if the result of the condition expression is true (non-zero), otherwise the program flow continues.

```cpp
if (condition) statementOne else statementTwo
```

StatementOne is executed if the result of the condition expression is true (non-zero), otherwise StatementTwo is executed.

The condition expression can use the following comparison operators, which return a `true` (or simply a non-zero value in older compilers) if the test is true, otherwise `false` (or zero): `== !!= < <= > >=`.

Logical operators are used to provide support for more sophisticated conditions:

- `!`, the NOT operator, negates its argument.
- `||`, the OR operator, only evaluates the second argument if the first is false (or zero).
- `&&`, the AND operator, only evaluates the second argument if the first is true (or non-zero). This is a useful property when testing for validity, as in this example using pointers:

```cpp
if (p && p->IsValid()) statementOne
```

This ensures that `p` is only dereferenced if it is non-zero, that is not `NULL`.

C++ also provides the conditional expression, or ternary operator `?:`, which is sometimes more convenient for simple selections:

```cpp
char* name = isClangerShort() ? "Small" : "The Major";
// which is the equivalent of
char* name = NULL;
if (isClangerShort())
    name = "Small";
```
else
    name = "The Major";

In Symbian OS C++ programming, using the conditional expression is discouraged, as it can lead to obscure code.

**switch (expression) statements**

The integral value of the expression determines which of the statements is (or are) executed. The `switch` statement is well suited for testing against sets of constants and enumerations rather than purely propositional evaluations.

```c++
// priority is a TPriority enumeration type
// See Section 1.1 for more information
char* statusString = "idle";
switch (priority)
{
    case EPriorityLow:
    statusString = "Low";
    break;
    case EPriorityStandard:
    statusString = "Standard";
    break;
    case EPriorityHigh:
    statusString = "High";
    break;
    case EPriorityUserInput: // deliberate fall through
default:
    statusString = "User Input";
}
```

The `priority` value is evaluated and the matching `case` is executed. If no `case` is matched, the `default` is executed. An important point to note is that unless each `case` statement is terminated with the `break` keyword, the execution will continue on to the next `case` (including `default`) regardless of the value of the expression.

Neither the `case` nor `default` keywords may be used outside the scope of the `switch` statement. The `break` keyword is also valid in loop and conditional statements.

**Iteration statements**

There are three iteration or loop statements in C++: `while`, `for` and `do`. All the statements repeat a block of instructions until some condition is met or a `break` is used.

```c++
while (condition) statement;
```
for (for-init-statement; condition; expression) statement;
do statement while (condition);

The while statement is typically used in operations such as reading an input stream where there are no obvious numeric limits, making a logical test more practical.

Token token = start(stream); // get a token from an input stream while (token != EOF) // not the end of the input stream {
    // ... do something
    token = next(stream); // and the next one
}

But it is entirely possible to achieve the same result with a for loop:

for (token=start(stream); token!=EOF; token=next(stream))
    {
        // ... do something
    }

The for statement has an advantage as its loop variable, condition and loop variable update expression are all on one line, which reduces the chance of error, but it can make code logically unclear. For this reason, for loops are best suited to regular order sequences:

// A regular kind of loop!
for (int ii=0; ii<10; ii++) // do something 10 times
    {
        // ... do something
    }

The do statement allows the code it is controlling to be executed once before the control condition is tested for. Thus, the precondition is that the loop code is safe for the first iteration. In the example below, the stream definitely contains tokens.

token = start(stream); // pre-condition: The stream is not empty
do {
    // ... do something
    token = next(stream);
} while (token!=EOF);

The break keyword may be used to stop the flow of execution and resume it after the loop. A more finely grained manipulation of loop
execution is the `continue` keyword, which stops the current iteration of the loop and starts the next iteration immediately:

```cpp
definition
for (token=start(stream); token!=EOF; token=next(stream))
{
    if (token==SPACE) continue; // ignore spaces, skip to next token
} // ... do something
while (Param p = getParam(stream))
{
    if (illegalParam(p) break; // drop out
    // ... do some param processing
} // Param p is out of scope
}
```

The above code segment reads a stream and ignores spaces by continuing to the next token. The inner `while` loop retrieves the token parameters and tests to see if they are legal. If not, the `while` loop is terminated and drops back to the outer `for` loop. Note the declaration of the `Param p` variable in the condition expression and its scope within the inner loop only.

**Exceptions and Notes**

- The One-Definition Rule (ODR) is not covered in any depth here; see [Stroustrup 2000 Section 9.2.3].
- There are a number of non-conditional ways to break out of a loop: `panic`, `leave` or simply `return`.

**Exam Essentials**

- Know the use and properties of the declaration and definition statements
- Recognize the use of the `extern` keyword
- Cite and understand initialization of variables and their scope
- Understand the purpose, syntax and behavior of C++ loop statements (`while`, `for` and `do`)
- Specify the behavior and effect of the `continue` and `break` keywords in a loop
- Specify the syntax and behavior of C++ conditional statements (`if` and `switch`)

**References**

[Stroustrup 2000 Sections 6.1 and 6.3 and Exercise 6.6]
1.3 Expressions and Operators

Critical Information

Operators

Operators are usually symbols, such as +, -, *, but can also be names, such as new or delete, and have a number of qualities aside from their actual purpose or function:

- An operator may be unary, binary or ternary (that is may take one, two or three arguments or operands).
- Operators have associativity.
- Unary operators may be prefix or postfix.

The number of operands an operator or function takes is sometimes referred to as arity. All operators return a single value. The table shows some examples:

<table>
<thead>
<tr>
<th>Type</th>
<th>Common Form*</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>operator(value)</td>
<td>n++</td>
</tr>
<tr>
<td></td>
<td>operator(expr)</td>
<td>--n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>!b</td>
</tr>
<tr>
<td>Binary</td>
<td>value operator value</td>
<td>a + b</td>
</tr>
<tr>
<td></td>
<td>value operator expr</td>
<td>a &gt;&gt; b</td>
</tr>
<tr>
<td></td>
<td>expr operator expr</td>
<td>a &amp;&amp; b</td>
</tr>
</tbody>
</table>

*This is pseudocode notation, not formal language definition syntax

Associativity refers to the order of evaluation.

- Left-to-right associativity: a+b+c means (a+b) +c (that is, a + b is evaluated first)
- Right-to-left associativity: a+b+c means a+ (b+c) (that is, b + c is evaluated first)

A great proportion of operators in C++ have left-to-right associativity. The increment and decrement operators (++ and --) have two forms: postfix and prefix.

- Prefix increment: a=++b means b=b+1; a=b;
- Postfix increment: a=b++ means a=b; b=b+1;
Operator precedence

There are priority rules for the order of evaluation of operators within an expression; that is, which operator is evaluated before another. For example, the multiplication operator * has a higher precedence than the plus operator +.

\( a + b \times c \) means \( a + (b \times c) \); that is, \( b \) is multiplied by \( c \) before \( a \) is added to the result.

This is called operator precedence and should not be confused with operator associativity, which is the order of evaluation of results for multiple instances of the same operator or operators of the same precedence.

In the following table, operators in the same box have the same order of precedence and the boxes are arranged with the highest precedence at the top of the table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>::</td>
<td>Scope resolution (global and class)</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>.</td>
<td>Member selection (object)</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Member selection (pointer)</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>[]</td>
<td>Subscript (array)</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>()</td>
<td>Function call</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>++</td>
<td>Postfix increment</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>--</td>
<td>Postfix decrement</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>static_cast</td>
<td>Type cast</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>sizeof</td>
<td>Size of object or type</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>++</td>
<td>Prefix increment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>--</td>
<td>Prefix decrement</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>~</td>
<td>One’s complement</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>!</td>
<td>Logical not</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>+</td>
<td>Unary plus</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>&amp;</td>
<td>Address-of</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>*</td>
<td>Indirection</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>new</td>
<td>Create object</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>delete</td>
<td>Delete object</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>()</td>
<td>Cast (cohesion)</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>.*</td>
<td>Pointer-to-member (objects)</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>-&gt;*</td>
<td>Pointer-to-member (pointers)</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>Operator</td>
<td>Name</td>
<td>Associativity</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>+</td>
<td>Plus</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Left shift</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>==</td>
<td>Equality</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive OR</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise inclusive OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e1?e2:e3</td>
<td>Conditional/ternary</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>*=</td>
<td>Multiplication assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>/=</td>
<td>Division assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>%=</td>
<td>Modulus assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>+=</td>
<td>Plus assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>-=</td>
<td>Subtraction assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>Left-shift assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>Right-shift assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Bitwise AND assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>Bitwise inclusive OR assignment</td>
</tr>
<tr>
<td>^=</td>
<td>Bitwise exclusive OR assignment</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>,</td>
<td>Comma</td>
<td>Left-to-right</td>
</tr>
</tbody>
</table>
Although memorization of the precedence order is not required for the ASD exam, it is important to be familiar enough to know that scope resolution, member selection, pointer operators and unary operators are towards the top, with arithmetic and conditional operators making up the middle and the lowest being the assignment operators. Note the low precedence of the logical AND and OR.

Exceptions and Notes

`a++ + a++` is undefined: the variable is read twice in the same expression, producing ambiguities during the evaluation.

Exam Essentials

- Specify the syntax and meaning of unary and binary operator expressions
- Understand the difference between precedence and associativity
- Recognize common operator categories including logical, prefix and postfix
- Demonstrate awareness of general operator precedence and associativity rules

Reference

[Stroustrup 2000 Sections 6.2.1–6.2.5, Sections 11.1 and 11.2, Section A.5]

1.4 Functions

This section deals with the syntax and semantics of non-member functions, that is functions that do not belong to a class.

Critical Information

Declaration

A function may be declared separately from its definition and declared more than once, provided the declarations are exactly the same. A function declaration is typically referred to as a function `prototype`. The clear separation between declaration and prototype enables a function to be used before it is defined.

For the compiler to ensure that a function is called legally, the function prototype needs to contain the function name, a list of parameter types and a return type. The parameter names are not required in the declaration.
Parameter lists in themselves are declarations containing the type name and an optional qualifier (`const`). As well as fundamental and user-defined types, pointers (`*`), references (`&`) and arrays (`[]`) can be passed.

A parameter may be initialized with a `default` value, so that when the function is called without a value for this parameter, the default value is assumed. The default arguments must always be at the end of the parameter list and can only ever be declared once, regardless of the number of times the function is declared.

```cpp
// an illustrative example
enum OperatorType {ADD, SUBTRACT /* , ... etc */ }; // Enum for selecting
int math (const int a, const int b, const OperatorType optype = ADD); // default add

// Definition - note the default value cannot be declared again
int math(const int a, const int b, const OperatorType optype /*=ADD*/) {
  switch (optype) { /* return something... */ } // the mechanics
}

// In use
int res = math(2,2); // res == 4
res = math(2,2,ADD); // res == 4
res = math(math(2,2,SUBTRACT),2); // res == 2
```

The `math` function takes two operands and a selection `enum`, which has the default value of `ADD` (not re-declared in the definition), and returns an `int`.

### Inline functions and macros

For simple and often-used functions, a C developer would use the `#define` macro directive to remove the overhead associated with function calls and improve efficiency.

```cpp
#define ADD(a,b) (a+b)
int res = ADD(2,2); // whenever ADD is encountered, it is substituted with (a+b)
```

Macros are generally discouraged in C++ due to lack of type checking and behavioral problems when the `#define` is more complex.

Matching of `if-else` statements does not behave as expected:

```cpp
// the if problem
#define TEST(A) if(cond) A;
int x = 0;
if (okay) TEST(x) else x = 42; // The if in the macro is matched to
                                // the else rather than the intended if(okay)
```
If a \#define contains a series or block of statements, there are unexpected results in loops:

```c
// block problem
#define BLOCK stmt1; stmt2; stmt3;
for (ii=0;ii<10;ii++) BLOCK; // Only stmt1 is executed inside the loop
  // stmt2 and stmt3 are executed outside
```

Pseudo-macro parameters do not behave as expected:

```c
// Multiple substitution
#define SQUARE(n) (n*n);
int n=2;
int res = SQUARE(n++);
// n is now 4 i.e. incremented twice
```

The \texttt{inline} keyword provides C++ developers with the same efficiency as a macro without the behavioral ramifications. The function body is substituted wherever the call is made.

```c
inline int square (int n)
{
    return n*n;
}
// ...
int n = 2;
int res = square(n++);
// n is 3 i.e. incremented once, the desired effect
```

The \texttt{inline} keyword is only a request to the compiler. It does not guarantee the substitution and it may still be treated as a normal function call. When a member function is fully defined in a class header file, in effect it becomes inline by default.

\textit{Passing an unspecified number of parameters}

To recap, a function prototype has a return type, a name and a parameter list. The parameter list contains the parameters that the function uses, and parameters at the end of the list may be initialized with a default value.

A function can also allow an \textit{unspecified} number of arguments to be passed to it. The following example shows the syntax for declaring and using such a function, and how to access the arguments within the function body.

```c
enum OperatorType {ADD, SUBTRACT};
// Declaration
int math (const OperatorType ty ...); // the ellipsis ... means any number
// Definition
int math (const OperatorType ty ...) {
    va_list ap; // param list
    va_start(ap,ty); // where to start in the list i.e. after ty
    int x = va_arg(ap,int); // go through the parameters, which are ints
    int res = 0;
    while (0!=x) // while not zero
    {
        switch (ty)
        {
            case ADD: // other cases, e.g. SUBTRACT, not used
                default:
                res+=x;
            }
            x = va_arg(ap,int); // Get next param
        }
    va_end(ap); // clean up the stack frame
    return res;
}

// Called
int r = math(ADD,1,2,3,4,5);

e32def.h contains the va_list, va_start and va_end macros.

The ellipsis (... ) tells the compiler that the number of arguments that follow the const OperatorType is unknown. Variable argument macros are provided to access the list:

- va_list is a pointer to the list of arguments.
- va_start tells the code where to start the list.
- va_arg gets the next parameter, supplying its type.
- va_end cleans up the stack.

**Passing arguments by value and by reference**

C++ provides two ways of passing arguments, by value and by reference. When an object is passed by value, a local copy is made on the function local stack and all subsequent modifications are made to that copy without any side-effect modification to the original copy. When a reference is passed, no local copy is made and all manipulations are carried out directly on the original object.

As passing by value actually copies the object, large values such as data structures can cause efficiency problems not only in taking up local stack space but in the action of copying. Passing by reference only copies the address, typically 4 bytes, and is therefore more efficient.

To prevent any unnecessary modifications to the referenced object, the const keyword is used to indicate that it will not be modified and is being passed as a reference for efficiency reasons.
There can be some confusion as to how the ampersand operator, \&, is used in declarations and expressions (see Section 1.2). In declarations of variables or functions, \& signifies a reference type, whereas in statements and expressions, such as when calling a function, \& signifies address-of.

In the example above, when function is called, the address of \textit{d} is passed (parameter ptrVal). Incidentally there is no concept of “pass by pointer”; a pointer is an address and the address is passed by value.

\textbf{Passing arrays to functions}

Passing arrays as arguments is a special case. Consider the following examples:

```c
void foo (int array[]) // pass an array
{
    array[0] = 42; // assign 42 to the first element
}
void bar (int* p) // pass a pointer to an int
{
    p[0] = 42; // assign 42 to the first element!?  
}
// Calling
int myArray[] = {1,2,3}; // an initialized 3 element array
foo(myArray); // pass the array
bar(myArray); // pass the array – the same way
```

In the above example \texttt{foo(int array[])} has the expected syntax for passing an array as argument. What in fact happens semantically is a little less obvious: as an array variable is always interpreted as a pointer to its first element (see Section 1.1), the parameter \texttt{int array[]} is converted to \texttt{int* array} when passed as function parameter. This is sometimes called array-to-pointer decay.

Thus \texttt{foo()} and \texttt{bar()} are semantically the same and interchangeable. An implication of array-to-pointer decay is that an array can never be passed by value. There is no local copy of \texttt{array[]}; in \texttt{foo()}, any modifications to \texttt{array} directly change \texttt{myArray}. 

```
void function (int val, // by value
    int& ref, // by modifiable reference
    const int& ReadOnlyRef, // by non-modifiable reference
    int* ptrVal) // by value
{
    val = 42; // val will only contain 42 for the scope of this function
    ref = 42; // The variable referenced is now 42
    ReadOnlyRef = 42; // *** Illegal *** attempt to modify a constant
    *ptrVal = 42;
}
// In use
int a = 0,b = 0,c = 0,d = 0;
function (a,b,c,&d); // post-conditions: a=0,b=42,c=0,d=42
```
The size of the array is not known to the function. To prevent out-of-range subscribes, that is, accesses to locations past the end of `array`, it is good practice either to pass the length as an additional parameter or to define a *termination* value for the last element.

The same rules for multidimensional array declarations apply to passing parameters: only the first dimension may be undefined. The size of the second and any subsequent dimensions must be known, or “bound”, before the assignment of values.

```cpp
void foobar (int array[][3])
{
    array[1][1] = 42;
}
int array[][3]= {(1,2,3),(1,2,3),(1,2,3)};
foobar(array);
```

The syntax for interchanging pointer notation with multidimensional arrays is tricky. `barfoo (int* array[3])` takes a single array of `int` pointers, whereas `barfoo (int (*array)[3])` takes the same effective parameter as `foobar (int array[][3])`, a multidimensional array of `ints`.

### Return values

A value must be returned from any function that is not declared `void` and, conversely, `void` functions cannot return a value. Only references and values (including pointers) may be returned. A function should never return a reference or pointer to a variable local to the function scope, that is on the stack.

```cpp
// bad pointer return
int* foo () // returns an int pointer
{
    int n = 42;
    return &n; // returns the address of n
} // n is out of scope, thus the return address is invalid

// bad reference return
int& bar () // return a reference to an int
{
    int n=42;
    return n; // the compiler creates a reference to n
} // n is out of scope, thus the reference is now invalid
```

The following code is an example of returning a reference safely, because it refers to a member variable which does not go out of scope:

```cpp
int& TTemperature:GetTemperature() {return iTemperature;}
```
Particular attention is required when returning modifiable references, because the object can then be modified by the calling code.

```c
TTemperature temperatureGauge;
int& temp = temperatureGauge.GetTemperature();
++temp; // the member variable TTemperature::iTemperature has
         // now been modified
```

**Pointer to function**

In the same way as pointers to variables, it is also possible to have pointers to functions. C++ allows the address of a function to be acquired in the same way as the address of a variable, by using the address-of operator, `&`.

```c
// simple function definition
void foo(int& n)
{
    n = 42;
}
// Function pointers
void (*bar)(int& n) = &foo; // the identifier bar is a pointer to a
// function whose prototype must contain the parameter
// list of a single ref to an int
// It is initialized with the address of foo()
int n = 0;
bar(n); // Call foo via bar
```

As function pointer declarations are very similar to actual functions, there are some syntax specifics that require careful attention. A pointer-to-function identifier must be put in brackets (`*bar`); this tells the compiler it is parsing a pointer to a function and not another function declaration or a data-pointer declaration. The return type (`void`) and the parameter list (`int& n`) indicate the prototype of the function the pointer will point to, that is (`*bar`) points to a function with the following prototype:

```c
void function-name (int& n);
```

For brevity of code, the use of the address-of operator, `&`, and the dereference operator, `*`, is optional.

```c
void (*bar)(int& n) = foo;
bar(n);
// is the same as
void (*bar)(int& n) = &foo;
(*bar)(n); // braces required to indicate not to dereference a return
// value
When passing parameters, function pointers behave in same way as other pointers.

// foo definition
void foo(int& n)
{
    n = 42;
}
// foobar definition with function pointer
void foobar (int& n, void(*callback)(int&n))
{
    callback(n);
}
// in use
int n = 0;
foobar(n,foo);   // n is now 42

Function pointer arguments are typically used in C as a method of dynamically selecting a function without knowing its name at compile time, thus enabling modifications to program behavior without unnecessary recompilation. For example, `foo()` could be completely rewritten without any modification to `foobar()`. In C++, virtual functions have superseded the need for function pointers, but it is still important to recognize the function pointer syntax. (For more on virtual functions, see Chapter 3.)

Exam Essentials

- Understand the syntax of a function prototype
- Cite the purpose of the `inline` keyword
- Understand the rules for passing default arguments and an unspecified number of arguments to a function
- Recognize value, reference, array and pointer parameter passing
- Understand the scope of function blocks and return by reference and value
- Specify the syntax for pointers to function assignments
- Recognize pointer to functions as callback parameter arguments

References

[Dewhurst 2005 Items 6, 14 and 25]
[Meyers 2005 Items 20, 21 and 28]
[Stroustrup 2000 Chapter 7 and Section C7.3]
1.5 Dynamic Memory Allocation

There are three types of storage scope: static, automatic and dynamic. Scope is defined by the minimum duration or lifetime of an object. Static and automatic storage occur when the objects are defined (see Section 1.2). Static objects last for the duration of the program and are indicated by the keyword static. Automatic objects only exist within the block where they are defined. The reserved word auto is now moribund in C++. This section describes dynamic storage allocation.

Critical Information

New and delete operators

Heap memory, or free storage, allocation and deallocation are provided by the operators new and delete. The new operator takes one argument, the size of the object type, and allocates the amount of memory required by that type, returning the location of the memory allocated.

```c++
int* p = new int; // allocate 4 bytes == sizeof(int);
char* pc = new char; // allocates 1 byte == sizeof(char);
//... p and pc are in scope until ...
delete p; // destroys the memory - frees 4 bytes
delete pc; // ... frees 1 byte
```

The duration of storage for p and pc is until delete is called to destroy them, thus they will exist outside the block they were defined in.

```c++
int* foo ()
{
    int* p = new int; // p variable contains the address of allocated heap
    return p; // returns by value.
} // p out of scope but the allocation is 'alive'
int* p = foo(); // address is now contained by p i.e. outside foo’s
    // function scope.
// do something ...
delete p; // now the memory allocated by foo is freed
```

For arrays, C++ provides an additional pair of operators, new[] and delete[]. Along with the type size, the number of elements to be allocated is also passed:

```c++
int* p = new int[10]; // 10*sizeof(int)
```
It is important that `delete []` (array delete) is called. Calling the `delete` operator only frees the first element and in doing so orphans the other nine elements (9*4=36 bytes memory leaked until the program ends).

**Placement new**

At its most basic, the `placement` `new` operator tells the compiler not to use the default `new` operator but another `implementation` or overload of `new`. If another overload is not explicitly declared by the developer, `placement` `new` is used to construct objects at a specific address passed to it — that is, either pre-allocated memory or memory-mapped hardware.

```cpp
unsigned char* heap = new unsigned char[1000]; // heap allocate
int* p = new (heap)int; // placement new
```

There is no equivalent `placement delete`, thus it is left to the developer to ensure that `p` is not deleted as no new memory was actually allocated. The `heap` memory must eventually be freed. `Placement new`, used like this, is limited — in the example below the `q` identifier points to exactly the same memory as `p`.

```cpp
int *q = new (heap)int; // q is now pointing at exactly the same memory location as p
// *** better
int* q = new (heap+sizeof(int))int;
```

This is messy; arguably a better approach would be to deal with the arithmetic inside the `new` operator itself. C++ allows developers to rewrite the global `new` operator, `void* operator new (size_t size)`, but, paraphrasing Stroustrup, “this would not be a task for the fainthearted.”

Symbian OS overloads the global operator `new`, to take a `TLeave` parameter, in addition to the implicit size parameter. The `TLeave` parameter is ignored by operator `new`, since it is only used to differentiate it. The implementation calls a version of the Symbian heap allocation function that causes a leave if there is insufficient memory for the allocation. See Chapter 5 for more information about Symbian OS leaves.

**Exam Essentials**

- Understand C++ free store allocation scope using the `new` and `delete` operators
- Recognize the syntax and purpose of `placement new`
1.6 Tool Chain Basics

This section deals very broadly with C++ tools and file organization, and serves as an introduction to the more detailed discussion of the Symbian OS tool chain in Chapter 14.

Critical Information

File and program structure

The smallest unit of compilation is a file. If only one character changes in that file then the whole file needs to be compiled again. Well-written programs are typically organized into a number of files containing source code that is organized in logical groups – usually based on the design architecture.

A C++ compiler has two distinct stages when it is passed a source file to build: macro preprocessor and compilation into object code.

Preprocessor

The preprocessor parses through the source file scanning for the directives. A directive starts with the hash sign, #.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include</td>
<td>Includes another file, usually header files *.h</td>
</tr>
<tr>
<td>#define</td>
<td>Allows the user to define a macro. e32def.h contains the most commonly used Symbian OS macros.</td>
</tr>
<tr>
<td>#ifdef</td>
<td>May be used to turn code on and off, typically in conditional compilation</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>#endif</td>
<td></td>
</tr>
</tbody>
</table>

The preprocessor will expand all the macros in the code. This also entails expanding the #include directive which adds the specified file to the current source file (see Section 1.4).
Separate compilation

The C++ language already provides the mechanism to separate variable and function declarations from definitions. To extend this feature and allow separate compilation, the #include directive allows the placing of declarations into separate files (*.h files) from their definitions (*.cpp). Thus, a third party may include the header and compile against it without the accompanying source definition file.

Any function definitions contained within a header file are automatically treated as inline, since the whole header file, including definitions, is effectively inserted into the including source file. Thus, if a definition is modified, all files that include it must be recompiled; otherwise different files may end up using different versions of the same function.

As function definitions – the code and algorithms – are more likely to change than their prototype declarations, it is recommended that inline functions be kept to a minimum and used only for very simple common functions (see Chapter 16 for more information).

Compiling and linking

After preprocessing, the compiler stage is executed. This is separated out into lexical analysis, syntax analysis, symbol-table generation and code generation. For the most part the source file is treated as a stream which is tokenized; a recursive syntax tree is built and validated and the target code is generated.

The compiler may be instructed to generate additional symbolic data binding the object code to the source code and associated symbols. This is used by a debugger to halt and step through the execution of the program.

Using #ifdef _DEBUG (typically) the developer may add debugging-specific information (for example log files) and checks including pre-condition or value assertions, for example:

```c
ASSERT(p!=NULL);
```

More sophisticated on-target debugging is also possible where code is executed on a native target device, usually a development board. Using a debugger stub on the target, commands and data may be sent back and forth to a PC via a communication port. An example is MetroTRK for Symbian OS.

Once the target code has been generated along with resources such as UI components (see Chapter 14), the object code is linked with the other compiled files in the program, the C++ libraries and any third-party libraries, including any extensions specific to an operating system (for example, S60).
At this stage, any inconsistency in declarations and definitions will become apparent, usually in the form of an unresolved external. All extern declarations are resolved during linkage. Finally, on successful linkage, the program is ready for execution.

Exam Essentials

• Understand the function of the C++ tool chain (for example compiler and linker)
• Recognize the lexical and syntax-parsing stages of compilation
• Be able to describe the purpose of the C++ preprocessor, specifying common directives
• Understand the role inline functions play in C++
• Know how to use the extern keyword

References

[Stroustrup 2000 Chapter 9 and Section 24.4]