1.5 مقدمة عن لغات البرمجة: *Programming Languages Introduction*

ينظر إلى الحاسب على أنه مكون من جزئين رئيسين:

- المكونات المادية *Hardware*.
- البرمجيات *Software*.

ولتسهيل الدراسة يتم تجزئة كل قسم على حده. فتقسم المكونات المادية إلى وحدات الإدخال *input units* ووحدات الإخراج *output units* ووحدة النظام *system unit* (المكونة من وحدة المعالجة المركزية و وحدة الذاكرة). وتقسم البرمجيات إلى أنظمة التشغيل ولغات البرمجة والبرامج التطبيقية. سنعطِ الآن فكرة عن لغات البرمجة المختلفة؛ بعد أن استعرضنا العديد من المكونات المادية، وخصّصنا فصلاً موجزاً للتعرف على أنظمة التشغيل.

2.5 لغات البرمجة: *Programming Languages*

تنقسم هذه اللغات بصفة عامة إلى نوعين هما:

- اللغات منخفضة المستوى *Low-Level Languages*.
- اللغات عالية المستوى *High-Level Language*.

هناك فارق كبير بين هذين المستويين في الإمكانيات وسهولة التعامل مع الحاسب، بالإضافة إلى سهولة تعلم اللغة وفهمها، وبما أن اللغات عالية المستوى تستخدم كلمات إنجليزية معينة ورموز رياضية مألوفة، فهي أسهل في تعلمها وفهمها وأمتع من حيث الاستخدام.

1.2.5 اللغات منخفضة المستوى: *Low-Level Languages*

تشمل هذه اللغات لغة الآلة *machine language*، ولغة التجميع *assembly language*. وفيما يلي نظرة سريعة على كل منهما.

لغة الآلة: *Machine Language*

تعتبر من أول اللغات ظهوراً، وهي اللغة الوحيدة التي يفهمها الحاسب مباشرة دون وسيط، ويجب أن تتحوّل إليها كل تعليمات البرامج المكتوبة باللغات الأخرى. وتتكون من رمزين هما: الصفر المنطقي والذي يعني غياب الجهد الكهربائي والواحد المنطقي والذي يعني تطبيق الجهد الكهربائي. تَجَمُّع وتسلسل هذين الرمزين يعبر عن الأوامر المختلفة والبيانات التي يتكون منها البرنامج. من أهم مساوئ هذه اللغة صعوبة تعلمها

خاصة أنّ لكل معالج لغة آلة خاصة به لذا فهي تتطلب معرفة واسعة في تصميمه وبنيته بالإضافة إلى صعوبة اكتشاف الأخطاء في برامجها، ممّا أدى إلى تطوير هذه اللغة إلى لغة التجميع.

لغة التجميع: *Assembly Language*

تعتمد هذه اللغة على الرموز المختزلة، أو المنمنمة *Mnemonic Code*، وهي اختصارات لكلمات ذات مدلول لغوي محدد مثل التعليمة *ADD* والتي تدل على الجمع *addition*، و *MOV* التي تدل على النقل *move* و*MUL* والتي تشير إلى عملية الضرب *multiplication* وهكذا، مما جعل تعلم هذه اللغة أسهل نسبياً من لغة الآلة، بالإضافة إلى سهولة اكتشاف الأخطاء وتصحيحها. يحتاج البرنامج في هذه اللغة إلى برنامج آخر يسمى بالمجمع *Assembler* كي يقوم بتجميعه وتحويله إلى لغة المعالج الطبيعية والتي هي لغة الآلة. مما يؤكد أن الحاسب لا يتعامل مباشرة إلا مع لغة الآلة. لهذه اللغة العديد من المحاسن منها إمكانية استغلال طاقة وإمكانيات المعالج وبشكل دقيق حيث تعجز اللغات عالية المستوى عن الاستفادة من هذه الإمكانيات. ومع ذلك تبقى هذه اللغة صعبة التعلم ولها عيوب من أبرزها ارتباطها بالآلة، بمعنى أن لكل مُعالج لغة تجميعية خاصة به.

بناءً على ما سبق نقول إن كتابة البرامج بلغات المستوى المنخفض تعتمد على معرفة واسعة بالتصميم الداخلي للحاسب (المعالجات، المقاطعات، مسارات البيانات، عناوين الذاكرة، ...الخ). مما جعل العلماء يفكرون بلغات تعزل المبرمج إلى حدٍ كبير نسبياً عن التصميم الداخلي للحاسب، عرفت هذه اللغات باللغات عالية المستوى.

2.2.5 اللغات عالية المستوى: *High-Level Languages*

تعتمد هذه اللغات على كلمات وعبارات إنجليزية واضحة المدلول مثل: (*write, read, input, print*)، وتُجنّب المبرمج مشقّة التعرف على التصميم الداخلي للمعالج، مما سهل عملية تعلّم هذه اللغات والإقبال عليها لحل المشاكل والتطبيقات العلمية والتجارية وغيرها. إلا أن تنفيذ برامج هذه اللغات يتطلب برنامجاً آخر يُدعى المترجم *compiler*.

من أهم مميّزات اللغات عالية المستوى:

- قابلية الحمل *portability* وهي تعني عدم ارتباط البرامج بمعالج معين مثل اللغات منخفضة المستوى بل يمكن تنفيذها على أي نوع من هذه المعالجات

وذلك لأنّ هذه اللغات مُصمّمة أساساً لحل نوعية محددة من المشاكل وليست لنوعية محددة من المعالجات.

- سهولة تعلمها وسهولة كتابة البرامج فيها، وذلك لاستخدامها كلمات وتعبيرات مشابهة لما يستخدمه الإنسان.
- سهولة اكتشاف الأخطاء وتصحيحها.
- توفير الجهد والوقت الذي كان يضيّعه المبرمجون عند كتابة البرامج بلغة الآلة أو لغة التجميع.

اللغات عالية المستوى كثيرة جداً، من أبرزها وأشهرها: *Fortran*، و*Cobol*، و*Basic*، و*Pascal*، و*C*، و*C++*، و*Java*، و*C#*، ... وغيرها.

تمايزت هذه اللغات فيما بينها من حيث القوة والسهولة فكانت لغة البيسك *Basic* اللغة الأكثر شهرة وشعبية، وشائعة الاستخدام بين جميع المبتدئين في البرمجة. وقد اشتقت اسمها من الحروف الأولى للعبارة الآتية:

Beginners All-Purpose Symbolic Instruction Code

والتي تعني: شيفرة التعليمة الرمزية متعددة الأغراض للمبتدئين. ظهرت هذه اللغة في ستينات القرن الماضي في الولايات المتحدة الأمريكية، ثم طورت من قبل معهد المعايير الأمريكية *ANSI* عام 1968م، ومن أهم إصداراتها *QBasic*.

بقيت هذه اللغات البرمجية بكافة أنواعها ضعيفة من حيث واجهات البرامج التي تنشئها، والواجهات المقبولة تتطلب كتابة آلاف الأسطر أثناء تصميم البرنامج، مما دفع الشركات لتطويرها إلى لغات مرئية *Visual*، وخصوصاً بعد ظهور نظام النوافذ *Windows* الذي يدعم البيئة الرسومية (*Graphic User Interface*). من اللغات المرئية: فيجول بيسك *Visual Basic*، *Delphi* (فيجول باسكال)، فيجول سي++ *Visual C++*، ...الخ.

وقد تبنت شركة مايكروسوفت لغة البرمجة *Qbasic* لتشكل نواة لغة بيسك المرئية (فيجول بيسك) *Visual Basic*، وقد ظهر أول إصدار لها عام 1991م وما يزال التحديث جارياً على هذه اللغة حتى الآن. وأصبحت لغة *Visual Basic* في إصداراتها الحديثة ذات قدرة على تصميم الأصناف *classes* والكائنات *Objects*، ومن هذه الإصدارات الحديثة *Visual Basic.Net*.

بناءً على ما سبق يمكن القول أن لغات البرمجة عالية المستوى تنقسم إلى:

- **لغات الأوامر السطرية:** تعتمد على كتابة الأوامر والتعليمات على شكل خطوات مرتبة ومنتهية، وهذه الخطوات تمثل ترجمة للخوارزمية، مثل: البيسك، والفورتران، والباسكال، وغيرها.
- **لغات مرئية:** تعتمد على الكائنات أو الأدوات والأحداث، حيث يتم الترابط بين الكائنات بعمليات برمجية، ثم يتم تنفيذ المشروع عن طريق الأحداث، ولذلك دعي أسلوب برمجة لغة بيسك المرئية بالبرمجة المقادة بالأحداث *Event-driven Programming* أي اللغة التي تنفذ تعليماتها وإجراءاتها عند اختيار الأحداث، والحدث هو كل تأثير يتم بالفأرة أو لوحة المفاتيح أو غيرها، مثل: النقر *Click* أو النقر المزدوج *Double click* وغيرها.

3.5 تطور أساليب البرمجة: *Evolution of Programming Techniques*

مرت أساليب البرمجة بثلاث مراحل:

- البرمجة العشوائية *Random Programming*.
- البرمجية الهيكلية *Structured Programming*
- البرمجة الشيئية *Object Oriented Programming*.

1.3.5 البرمجة العشوائية: *Random Programming*

يركز هذا الأسلوب من البرمجة على حل المسألة برمجياً وتحقيق الهدف دون النظر إلى عملية تنظيم البرنامج، وفي هذه الحالة يعاني البرنامج من صعوبة في التطوير، ومن صعوبة في اكتشاف الأخطاء، وربما حصل تكرار في بعض المقاطع البرمجية، كما أن استخدام الأمر *Go to* بكثرة يعيق فهم البرنامج وصعوبة تتبع خطوات التنفيذ ومن هنا جاء تعبير البرمجة العشوائية، وفي هذه الحالة ينظر للبرنامج على أنه كتلة واحدة. وهذا الأسلوب لا تظهر مشاكله إلا إذا كان البرنامج كبيراً وضخماً، أما في حالة البرامج التدريبية البسيطة فربما يكون مناسباً، لأنه لا يحتاج إلى تجزئة. سنقدم مثالاً يناسب هذا الأسلوب من البرمجة وآخر لا يناسب هذا الأسلوب.

مثال (1): يناسب هذا الأسلوب.

هذا البرنامج يقوم بإدخال عددين مختلفين وطباعة أكبرهما.

```
Input a,b
While a=b
        Input a,b
Wend
If a>b then print a else print b
```

مثال (2): لا يناسب هذا الأسلوب بسبب التكرار.

هذا البرنامج يقوم بإيجاد عدد التوافيق وفق العلاقة التالية:

$$F(n,r)=\frac{n!}{r!\times(n-r)!}$$

```
Input n,r                    For I=1 to r
While n<r                    w=w*I
        Input n,r            Next I
Wend                         For I=1 to n-r
s=1:w=1:v=1                  v=v*I
For I=1 to n                 Next I
s=s*I                        F=s/(w*v)
Next I                       Print F
```

2.3.5 البرمجة الهيكلية: *Structured Programming*

يعتمد هذا الأسلوب من البرمجة على تجزئة البرنامج إلى عدة برامج فرعية، بحيث يتم الربط بين هذه البرامج الفرعية لتشكل البرنامج العام. والمقصود بالبرامج الفرعية الدوال *functions* أو الإجرائيات *procedures*. يحتاج هذا الأسلوب إلى تخطيط جيد، وتظهر فاعليته في المسائل متوسطة الحجم والتي تطرح على الطلاب في المرحلة الجامعية، كما يُسهّل من اكتشاف الأخطاء، وإجراء عمليات التطوير، وعدم تكرار المقاطع البرمجية.

مثال (1): حساب عدد التوافيق وفق العلاقة السابقة بأسلوب البرمجة الهيكلية.

```
Input n,r                    Function G(m)
While n<r                    G=1
Input n,r                    For I=1 to m
Wend                         G=G*I
S=G(n)/( G(r)*G(n-r) )       Next I
                             End function
```

مثال (2): يناسب أسلوب البرمجة الهيكلية.

هذا البرنامج يقوم بطباعة جميع الأعداد الأولية من 2 حتى العدد المدخل *n*. العدد الأولي هو الذي لا يقبل القسمة إلا على نفسه أو الواحد.

Input n *While n<2* *Input n* *Wend* *For I=2 to n* *If Prime(I) =1 then print I* *Next I*	*Function Prime(m)* *Prime=1* *For I=2 to m-1* *If m mod I =0 then prime=0* *Next I* *End function*

4.5 تاريخ لغة C: *History of C*

تطوّرت لغة *C* من لغتين سابقتين هما *BCPL* و*B.BCPL* في عام *1967* من قبل *Martin Richards* كلغة لكتابة برامج أنظمة التشغيل والمترجمات. نمذج العالم *Ken Thompson* ميزات عديدة في لغته *B* بعد أن استخدمتها *BCPL* بعد استخدامها عام *1970* لإنشاء الإصدارات الأولية لنظام التشغيل *UNIX* في مختبرات *Bell* على الحاسب *DEC PDP -7*.

طورت لغة *C* بالأساس من لغة *B* عن طريق *Dennis Ritchie* في مختبرات *Bell* ونفذت بالأساس على حاسب *DEC PDP-11* في *1972*. تستخدم لغة *C* العديد من المفاهيم الهامة للغة *BCPL* و*B* بينما أضافت طباعة البيانات وصفات قوية أخرى للغة. عرفت لغة *C* كلغة لتطوير نظام التشغيل *UNIX*، أمّا اليوم فإنّ كل نظم التشغيل الرئيسة الحديثة تكتب في *C* و*C++*. اللغة *C* متوفرة لمعظم الحواسب وهي أيضاً مستقلة عن الكيان الصلب*hardware independent*. بالتصميم الحذر يمكن كتابة برامج *C* القابلة للحمل *portable* لمعظم الحواسب.

بنهاية السبعينيات طورت لغة *C* إلى ما يعرف اليوم بـ *"C* التقليدية *Traditional C"* وبعد صدور الكتاب *" The C programming language"* للعالمين *Kernighan* و*Ritchie* في عام *1978* فقد تم التركيز على هذه اللغة وأصبح هذا الكتاب من أشهر الكتب الناجحة التي تتعلق بعلم الحاسب على مر الزمن.

النمو السريع للغة *C* عبر الأنواع المختلفة للحواسب (المسماة أحياناً أرصفة الكيان الصلب *hardware platforms*) أدى إلى تغييرات عديدة كانت متشابهة لكنها غير

متوافقة. وكان هذا أمراً معضلاً لمطوري البرامج الذي يرغبون بتطوير شيفرة تنفذ على أرصفة مختلفة ومتعددة. أصبح واضحاً أن الإصدار المعياري لـ *C* أصبح حاجة ماسة. في عام *1989* صُدّق على أول معيار لهذه اللغة ثم طُوِّر عام *1999*.

1.4.5 مكتبة *C* المعيارية: *C Standard Library*

تتكون برامج لغة *C* من وحدات أو قطع تدعى الدوال *functions* وبالإمكان برمجة كل الدوال التي نحتاجها لتشكيل برنامج *C*، لكن معظم مبرمجي لغة *C* يستفيدون من الدوال الموجودة و التي تعرف بمكتبة *C* المعيارية *C standard library*. لذلك فإن هنالك جزأين لكي نتعلم البرمجة بلغة *C*. الأول هو تعلم لغة *C* نفسها والثاني هو تعلم كيفية استخدام الدوال الموجودة في مكتبة *C* المعيارية.

تستخدم عند البرمجة في لغة *C* كل من وحدات البناء التالية:

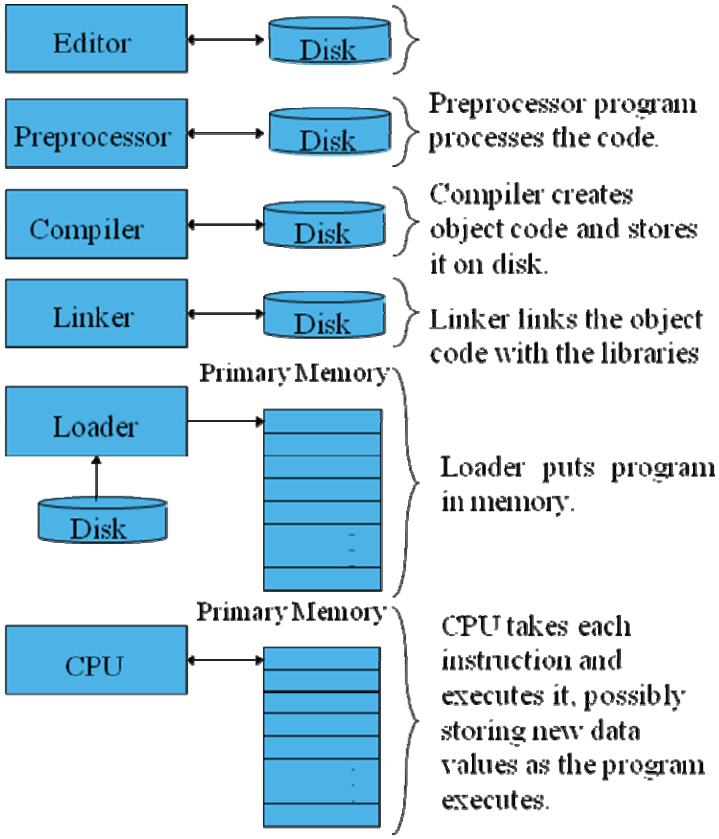
- دوال مكتبة *C* المعيارية.
- الدوال التي تنشئها بنفسك.
- الدوال المنشأة من قِبل أشخاص آخرين.

ميزة إنشاء دوال خاصة بك هي أنك ستعرف بالضبط كيفية عملها وستكون قادراً على تفحص شيفرة لغة *C*. وسيئتها هي الزمن المستهلك في تصميم وتطوير دوال جديدة.

2.4.5 أساسيات بيئة تطوير برنامج C نموذجي:

Basics of a Typical C Program Development Environment

تتكون أنظمة *C* بشكل عام من عدة أجزاء هي: بيئة تطوير البرنامج، واللغة، ومكتبة *C* المعيارية. توضّح المناقشة التالية بيئة تطوير *C* النموذجية المبنية بالشكل التالي:



تمر عادة برامج لغة *C* عبر ستة أطوار حتى يتم تنفيذها ، هذه الأطوار هي:

- الكتابة أو التحرير *editing*.
- المعالجة الأوليّة *preprocessing*.
- الترجمة *compiling*.
- الربط *linking*.
- التحميل *loading*.
- التنفيذ *executing*.

يتكون الطور الأول من عملية تحرير الملف وهذا يتم ببرنامج تحرير *editor program*.

سوف نفترض أنّ القارئ يعرف كيف يُحرّر برنامج ما. يقوم المبرمج بكتابة برنامج *C* بالمحرر ويجري التصحيحات الضرورية ثم يخزن البرنامج على وحدات التخزين الثانوية كالأقراص. يجب أن تنتهي أسماء ملفات *C* بالامتداد (*.c*).

يقوم المبرمج بعدئذٍ بإعطاء أمر لترجمة البرنامج، يقوم المترجم بترجمة البرنامج إلى شيفرة لغة الآلة (تعرف أيضاً باسم الشيفرة الهدف *object code*).

في نظام *C* يتم تنفيذ برنامج ما من قِبل المعالج بشكل تلقائي قبل بدء طور الترجمة من قبل المترجم. يستجيب معالج *C* الأولي لأوامر خاصة تدعى موجهات المعالج الأولي *preprocessor directives* والتي تُبيّن أنه يجب تنفيذ عمليات محددة على البرنامج قبل الترجمة. تتكون هذه العمليات عادة من إدخال ملفات أخرى في الملف كي تتم ترجمته كما تقوم بتنفيذ عمليات إحلال نصية مختلفة. يتم استدعاء المعالج الأولي بشكل تلقائي من قبل المترجم قبل أن يحول البرنامج إلى لغة الآلة.

يدعى الطور التالي بالربط *linking*. تحتوي برامج *C* عادة على مراجع لدوال معرفة في أمكنة مختلفة مثل تلك الموجودة في المكتبات المعيارية أو في المكتبات الخاصة لمجموعات من المبرمجين الذين يعملون على مشروع محدد *particular project*. تحتوي الشيفرة الهدف الناتجة من مترجم لغة *C* عادة على فراغات أو ثقوب *"holes"* بسبب هذه الأجزاء المفقودة. يقوم الرابط *linker* بوصل الشيفرة الهدف بشيفرة الدوال المفقودة لإنتاج صورة قابلة للتنفيذ *executable image* (بدون قطع ضائعة أو مفقودة).

يدعى الطور التالي بالتحميل *loading*. قبل تنفيذ برنامج ما فإنه يجب أن يوضع في الذاكرة وهذا يتم عن طريق المحمل *loader* والذي يأخذ الصورة القابلة للتنفيذ من القرص وينقلها إلى الذاكرة. كما يتم تحميل مكونات إضافية من المكتبات المشتركة التي تدعم البرنامج.

وأخيراً يقوم البرنامج تحت إشراف وحدة معالجة المركزية بتنفيذ البرنامج تعليمة تعليمة. قد لا تعمل البرامج من أول محاولة ويمكن لكل من الأطوار السابقة أن يفشل بسبب الأخطاء المختلفة التي سنناقشها.

مثلاً يمكن أثناء تنفيذ برنامج أن تحدث القسمة على صفر (وهي عملية غير صحيحة على الحواسب تماماً كما في الرياضيات). مما يجعل الحاسب يطبع رسالة خطأ. يجب عندئذٍ أن يعود المبرمج إلى طور التحرير ليقوم بالتعديلات والتصحيحات الضرورية و يتابع عبر الأطوار المختلفة مرة ثانية حتى يتم تنفيذ البرنامج بشكل صحيح.

معظم البرامج في لغة *C* تُدخل أو تُخرج البيانات. تأخذ بعض الدوال المحددة في *C* دخلها من الـ *stdin* (تدفق الإدخال المعياري *standard input stream*) والذي هو عادة لوحة

المفاتيح لكن يمكن أن يكون *stdin* متصلاً مع تدفقٍ آخر. تخرج البيانات عادة إلى *stdout* (تدفق الإخراج المعياري *standard output stream*) والذي هو عادة شاشة الحاسب لكن يمكن لـ *stdout* أن تكون متصلة مع تدفقات أخرى. عندما نقول أن برنامجا ما يطبع نتيجة فإننا نقصد أن النتيجة ظاهرة على الشاشة. يمكن للبيانات أن تخرج على أجهزة مثل الأقراص والطابعات. أيضا هناك تدفق يدعى تدفق الخطأ المعياري، ويشار إليه بـ *stderr*. يستخدم هذا التدفق (يتصل عادة مع الشاشة) لإظهار رسائل الخطأ. ويمكن توجيه بيانات الخرج النظامية *stdout* إلى جهاز غير الشاشة في حين يكون *stderr* مخصصا للشاشة بحيث يرى المستخدم الأخطاء وبشكل آني.

ولنوظّف الآن هذه المفاهيم في كتابة برامج بسيطة جداً تُوضّح هيكلة ومكونات البرنامج بلغة *C*.

3.4.5 برنامج *C* بسيط: طباعة سطر من نص

A Simple C Program: Printing a Line of Text

تستخدم لغة *C* رموزاً و تراكيب قد تبدو غريبة للشخص غير المبرمج للحاسب. يقوم البرنامج التالي بطباعة سطر من النص على الشاشة، وهو مبين بالشكل التالي:

```
1
2       /*   A First Program in C   */
3       #include <stdio.h>
4
5       /*function main begins program execution */
6       int main( void )
7       {
8          printf( "Welcome to C!\n" );
9
10         return 0; /* indicates that program ended successfully*/
11
12      } /* end of function main*/
```

خرج البرنامج

```
Welcome to C!
```

البرنامج الأول في لغة *C*

مع بساطة هذا البرنامج إلا أنه يبين ميزات هامه للغة *C*. لندرس كل سطر من البرنامج بالتفصيل.

السطر الثاني هو:

/ A First Program in C */*

يبدأ بـ */ وينتهي بـ */ هذا يعني أن هذا السطر هو تعليق *comment*. يدخل المبرمجون التعليقات *comments* في البرامج بهدف توثيقها و تحسين قابلية قراءتها *readability*. لا تؤثر التعليقات على الحاسب، فهو لا يقوم بتنفيذ أي حدث عندما يمر عليها، كما أن مترجم لغة *C* يتجاهل هذه التعليقات ولا يولد أي شيفرة هدف *object code* بالمقابل.

يبين التعليق السابق ببساطة الغرض من البرنامج، تساعد أيضاً التعليقات الأشخاص الآخرين في قراءة البرنامج وفهمه، لكن كثرتها قد تعقد عملية القراءة.

ملاحظة:

- يعتبر نسيان كتابة */ في نهاية التعليق خطأً برمجياً شائعاً. كما يعتبر بدء التعليق /* و إنهاؤه بـ */ خطأ برمجياً شائعاً.

السطر الثالث:

#include <stdio.h>

يعتبر مُوجِّهاً *directive* لمعالج *C* الأولي *preprocessor*. تعالج الأسطر التي تبدأ بالحرف # من قبل المعالج الأولي قبل ترجمة البرنامج. يخبر هذا السطر المعالج الأولي بأن يُضمّن *include* محتويات ترويسة الإدخال/إخراج القياسية *standard input/output header* المسماة اختصاراً *stdio.h*.

تحتوي هذه الترويسة في البرنامج على المعلومات المستخدمة بواسطة المترجم عند استدعاء دوال مكتبة الإدخال/إخراج القياسية مثل *printf*.

السطر السادس:

int main

يُشكّل هذا السطر جزءاً أساسياً من كل برامج لغة *C*. تشير الأقواس بعد كلمة *main* إلى أن *main* هي وحدة بناء برنامج تدعى دالة *function*. تحوي برامج لغة *C* دالة واحدة أو أكثر، ويجب أن تكون إحداها رئيسة *main* حيث يبدأ منها تنفيذ البرنامج في لغة *C*.

يجب أن يبدأ القوس الكبير الأيسر { جسم كل دالة (السطر رقم 7) والقوس المقابل الأيمن } يجب أن ينهي كل دالة (السطر 12). يطلق على هذين الزوجين من الأقواس مع جزء البرنامج المحصور بينهما اسم صندوق أو وحدة *block*. يعتبر الصندوق وحدة هامة من البرنامج في لغة *C*.

السطر الثامن:

```
printf ("Welcome to C!\n");
```

يُعلم الحاسب بأن يقوم بتنفيذ حدث ما وهو طباعة المحارف ما بين علامتي التنصيص على الشاشة. تدعى السلسلة *string* أحياناً باسم سلسلة المحرف *character string* أو رسالة *message* أو محارف ثابتة *literal characters*. كامل السطر بما فيه *printf* ووسائطها *arguments* بين الأقواس والفاصلة المنقوطة (;) تدعى جملة *statement*. يجب أن تنتهي كل جملة بفاصلة منقوطة (تعرف أيضا باسم منهي الجملة *statement terminator*).

عند تنفيذ الجملة *printf* تتم طباعة الرسالة *Welcome to C!* على الشاشة حيث تظهر المحارف على الشاشة بنفس الشكل الذي تظهر به تماما بين علامتي التنصيص في جملة *printf*.

لاحظ أنّه لا يتم طباعة المحرفان *n*\ على الشاشة. يدعى المحرف (\) محرف الهروب *escape character* وهو يشير إلى أن الأمر *printf* يجب أن يفعل شيئا ما غير عادي. عند مصادفة هذا المحرف في سلسلة ينظر المترجم إلى المحرف التالي ويرفقه مع هذا المحرف (\) ليشكل تتابع هروب *escape sequence*. يشير تتابع الهروب *n*\ إلى سطر جديد *new line*. عند ظهور سطر جديد في طرف السلسلة بواسطة *printf* فإنّه ينقل المؤشر الضوئي *cursor* إلى بداية السطر التالي على الشاشة.

يُبيّن الجدول التالي بعض تتابعات الهروب الشائعة:

تتابع الهروب	الوصف
\n	سطر جديد، ضع المؤشر الضوئي في بداية سطر جديد.
\t	مسافة أفقية، حرك المؤشر الضوئي إلى بداية *tab* أخرى تالية.
\a	تنبيه، دع نظام الحاسب يصدر صوتا.
\\	عارضة خلفية مائلة *back slash*، ادخل محرف \ في سلسلة.
\"	تنصيص مزدوج *double quote*، ادخل محرف تنصيص مزدوج في سلسلة.

قد يبدو آخر تتابعي هروب في الجدول السابق غريبين لأنّ المحرف \ له معنى خاص في السلسلة، بمعنى أن المترجم يتعرف عليه كمحرف هروب، لذلك فإننا نستخدم محرفين متتالين منه (\\) لوضع وإدخال واحد في السلسلة. تعتبر عملية طباعة محرف تنصيص مزدوج أيضا مشكلة لأنها تشير إلى حد السلسلة (بدايتها أو نهايتها) وهي في الواقع لا تطبع، لكن باستخدامنا لتتابع الهروب \" في سلسلة فإننا نشير إلى أن *printf* سوف يظهر علامة التنصيص العلوية في السلسلة.

السطر رقم *10*:

```
return 0; /* indicate that program ended successfully */
```

يدخل هذا السطر في نهاية كل دالة رئيسة *main*. الكلمة المفتاحية *return* هي إحدى وسائل عدة سنستخدمها للخروج من الدالة *exit a function*. عندما تستخدم جملة *return* في نهاية الدالة *main* كما هو واقع الحال في البرنامج السابق، فان القيمة *0* تشير إلى انتهاء البرنامج بشكل ناجح.

يبين القوس الكبير الأيمن { في السطر *12* بلوغ نهاية الدالة *main*.

ملحوظة:

- أضف تعليق للسطر الذي يحوي القوس الأيمن { والذي يغلق كل دالة بما فيها *main*.
- قلنا أن *printf* تجعل الحاسب ينفذ حدثا ما وعند تنفيذ أي برنامج فإنه يمكن اتخاذ قراراتٍ وأفعالٍ مختلفة فيه.

من المهم ملاحظة أنّ دوال المكتبة المعيارية مثل *printf* و*scanf* ليست جزءاً من لغة البرمجة. مثلا لا يمكن للمترجم إيجاد خطأ إملائي في *printf* و*scanf*. عندما يقوم المترجم بترجمة جملة *printf* فإنّه يُقدّم فقط فراغا في البرنامج الهدف لاستدعاء دالة من المكتبة. لكن لا يعرف المترجم مكان وجود دوال المكتبة هذه و الرابط *linker* هو الذي يعرف. عند عمل الرابط فإنه يقوم بإيجاد *locates* دوال المكتبة ويدخل الاستدعاء المناسب لدوال المكتبة هذه في البرنامج الهدف، وعندها يصبح البرنامج الهدف مكتملاً وجاهزاً للتنفيذ. في الواقع يُدعى البرنامج المترابط *linked program* بالبرنامج القابل للتنفيذ *executable*. إذا تم كتابة اسم الدالة بصورة خاطئة فإن البرنامج الرابط هو الذي يشير إلى الخطأ لأنه لن يكون قادراً على ملائمة الاسم في البرنامج في لغة *C* مع أيٍ من الأسماء المعرفة في المكتبات.

يمكن استخدام مفتاح *tab* في محاذاة المسافات لكنه قد يقف عند مسافات مختلفة. يُوصى بإدخال ثلاث مسافات أو مسافة بمقدار *1/4 inch* أي ما يعادل *tab*.

يمكن أن تطبع دالة *printf* الرسالة *Welcome to C!* بعدة طرق مختلفة.

يُقدّم البرنامج التالي نفس الخرج مثل البرنامج الأول، وهذا يتم لأنّ كل *printf* تواصل الطباعة من نفس المكان الذي توقفت عنده *printf* السابقة.

```
1
2     /*  Printing on one line with two printf statements*/
3     #include <stdio.h>
4
5     /* function main begins program execution*/
6     int main()
7     {
8       printf( "Welcome " );
9       printf( "to C!\n" );
10
11      return 0; /* indicate that program ended successfully */
12
13    } /* end function main */
```

خرج البرنامج

```
Welcome to C!
```

الطباعة على سطر واحد باستخدام جمل *printf* منفصلة

تطبع أول *printf* في السطر الثامن الكلمة *Welcome* وتُلحقها بفراغ، وتبدأ *printf* الثانية في السطر التاسع الطباعة من نفس السطر بعد الفراغ مباشرة.

يمكن لـ *printf* واحدة أن تطبع عدة أسطر باستخدام محارف سطر جديد إضافية كما هو مبين في المثال التالي:

```
1
2    /* printing multiple lines with a single printf */
3    #include <stdio.h>
4
5    /* function main begins program execution */
6    int main()
7    {
8       printf( "Welcome\n" );
9       printf( "to C!\n" );
10
11      return 0; /* indicate that program ended successfully */
12
13   } /* end function main */
```

خرج البرنامج

```
Welcome
to C!
```

الطباعة على عدة أسطر بجملة *printf* واحدة

في كل مرة يُصادف فيها تتابع الهروب *n*\ (سطر جديد *new line*) ينتقل الخرج ليستمر في بداية السطر التالي.

5.5 البرمجة الموجهة نحو الأشياء: *Object Oriented Programming*

تدعى أيضاً بالبرمجة الشيئية، أو الكائنية وهي البرمجة التي تحاكي الواقع. يعتمد هذا الأسلوب من البرمجة على بناء الكائنات التي تضم البيانات والإجراءات، وجملة ترابط الكائنات تشكل المشروع.

ظهرت البرمجة الشيئية *OOP* في بداية السبعينات من القرن الماضي، حيث بدت الحاجة ماسة لها، بعد أن واجهت البرمجة الهيكلية (الإجرائية) عدة مشاكل منها: البيانات غير المحمية، عدم القدرة على محاكاة الواقع، صعوبة تقسيم البرنامج إلى إجرائيات، عدم القدرة على إعادة الاستعمال، وغيرها من الأسباب.

قد يتصور البعض أن البرمجة الشيئية هي لغة برمجية جديدة، ولكن الأمر ليس كذلك، إنما هي أسلوب تنظيمي في البرمجة بكافة توجهاتها وعلى اختلاف لغاتها، وأن استخدام أسلوب البرمجة الشيئية لا يعني نسف كل ما تعلمناه كمبرمجين، لا بل هو الإلمام بالإمكانات المتاحة من خلال هذا الأسلوب.

اعتبر مصممو البرمجة الشيئية أن الأجسام من حولنا ما هي إلا كائنات (أشياء) تتفاعل مع بعضها وفق علاقات تتفق مع طبيعتها، بغض النظر عن كون هذه الكائنات حية أم جامدة. والهدف من ذلك هو مساعدة المبرمجين على كتابة برامج قابلة للتطوير وبمدة زمنية أقل.

مر معنا ذكر الكائن أو الشيء *Object*، فما هو الكائن وفق ما تراه البرمجة غرضية التوجه.

الكائن: *Object*

الكائن هو نمط جديد (متغير مركب يشبه نسبياً السجل في تعريفه البرمجي) ينتج من دمج البيانات *data* والإجرائيات *procedures* التي تعمل على تلك البيانات في كينونة واحدة.

الآن، عندما تحاول حل مشكلة في البرمجة الشيئية، لن تتساءل عن كيفية تقسيم المشكلة إلى إجرائيات أو دوال، بل عن كيفية تقسيمها إلى كائنات. التفكير بالكائنات بدلاً من الإجراءات له تأثير مفاجئ عن مدى سهولة تصميم البرنامج بهذا الأسلوب، وذلك بسبب المطابقة القوية بين الكائنات في المفهوم البرمجي والكائنات في الحياة الفعلية.

ما هي أنواع الكائنات التي تصبح كائنات في البرامج الشيئية؟ الجواب محصور عند حدود التصور فقط، لكن لنرى بعض الأمثلة:

- **كائنات بشرية:** تلاميذ، موظفين، زبائن، مندوبي مبيعات.
- **مكونات في ألعاب الحاسب:** الأشباح في لعبة المتاهة، الأحجار في لعبة الشطرنج، السيارات في لعبة سباق.
- **العناصر في بيئة نظام التشغيل:** الأطر والقوائم، مربعات الحوار وأزرار الأوامر، والكائنات الرسومية (الخطوط والمستطيلات والدوائر).

- **الكائنات المادية:** المصاعد في برنامج للتحكم بالمصعد، السيارات في برنامج حركة السير، الطيارات في نظام النقل الجوي.

العناصر الرئيسة في اللغات غرضية التوجه:

عند الحديث عن اللغات غرضية التوجه فلا بد من معرفة ثلاثة مفاهيم:

- التغليف *Encapsulation*
- الوراثة *Inheritance*
- تعدد الأشكال *Polymorphism*

أولاً: التغليف *Encapsulation*

الفكرة الأساسية للغات البرمجة الشيئية هي دمج البيانات والإجرائيات التي تعمل على تلك البيانات للحصول على نمط جديد من هذه البيانات، ألا وهو الكائن *Object*. عملية الدمج أو كيفية الدمج هذه، للحصول على الكائن تسمى تغليفاً.

مثال:

الكائن إنسان له اسم وعمر وطول، ...الخ. هذه بيانات *data* وله وظائف: يقرأ ويكتب ويسمع ويمشي...الخ وكل هذه إجراءات. عملية الدمج للبيانات والإجراءات في تعريف برمجي تُدعى تغليفاً. شكل وآلية التغليف يعود إلى اللغة المدروسة.

ثانياً: الوراثة: *Inheritance*

تعتمد نظرية البرمجة الشيئية على حقيقة وجود علاقة هرمية *hierarchy* وراثية بين عناصر البرنامج، حيث تأخذ العناصر الدنيا في هذه السلسلة الهرمية خواص العناصر العليا.

إذاً، الوراثة هي تشكيل بنية هرمية *hierarchy* تنازلية من الكائنات، حيث يرث كل كائن في البنية الهرمية إمكانية النفاذ إلى البيانات والإجرائيات في السلسلة التي تعلوه في البنية.

لتمثيل هذه العلاقة في البرنامج يمكننا تعريف بنى معطيات مركبة تدعى بالأصناف *Classes*. يدعى فيها العنصر الذي يقع تحت عنصر آخر في السلسلة الهرمية مشتقاً من العنصر الأعلى وفي المقابل العنصر الذي يقع فوق عنصر آخر في السلسلة يدعى أساساً

أو قاعدة للعنصر الأخفض، في الحقيقة فإن العنصر المشتق هو حالة خاصة من العنصر الأساس، حيث يملك كل خواص العنصر الأساس بالإضافة إلى خواصه الإضافية.

يمكننا بناء مثل هذه العلاقات مباشرة في البرمجة الشيئية وذلك باستخدام الوراثة، حيث تمكّن الوراثة الكائن المشتق من امتلاك كل خواص الكائن الأساس. وهذه الخاصية تجعل البرمجة أسهل إنشاءً وأكثر إمتاعاً، إذ لم نعد بحاجة إلى تعريف الخواص بالنسبة لكل عنصر، لأن العنصر يمكن أن يرث خواصاً من عنصر آخر.

مثال: الصنف إنسان يمكن قسمته إلى مجموعات تبعا للمهنة مثل: أطباء، معلمين، مهندسين، ...

الأطباء: قلبية، عينية، أذنيه، ...الخ.

المعلمون: رياضيات، فيزياء، تربية إسلامية، ...الخ.

المهندسون: مدني، عمارة، الكترونيات، حاسب...الخ.

وبنفس الطريقة:

معلم رياضيات: جبر، هندسة، إحصاء، ...الخ.

معلم لغة عربية: نحو، أدب، بلاغة، ... الخ.

وهكذا يمكن بناء شكل هرمي كبير يحقق ما ذكرناه سابقاً.

ثالثاً: تعدد الأشكال: *Polymorphism*

يقصد بتعدد الأشكال إعطاء أحد الإجرائيات اسما مشتركاً بين جميع الكائنات عبر البنية الهرمية، ويقوم كل كائن باستخدام هذه الإجرائية بطريقة خاصة به ومناسبة له.

ويوجد في حياتنا الواقعية مجموعة من الأنواع المختلفة والتي تتصرف بطرق مختلفة رغم إعطائها تعليمات متطابقة.

مثال: لنأخذ طلاب الجامعة: في كلية الطب، والهندسة، والآداب، والعلوم، ... ، ولنفرض أن إدارة الجامعة تريد إرسال استمارة تسجيل لكل طالب. فهل تبعث إدارة الجامعة استمارات مختلفة، لأن الطلاب ينتمون إلى فروع مختلفة؟ لا، بل تبعث استمارة لها شكل واحد والطلاب يعرفون كيفية تعبئة استمارة التسجيل الخاصة بهم.

6.5 لغة C++: *C++ Language*

تتفوق لغة $C++$ على لغة C بعدة خصائص ومميزات وتقدم إمكانيات البرمجة المُوجهة نحو الأشياء (*OOP*) *object-oriented programming* والتي تبشّر بمردود متزايد لحجم البرامج وجودتها وقابلية إعادة استخدامها *reusability*.

لم يتوقع مصممو لغة C الأوائل أن هذه اللغة ستصبح ظاهرة *phenomenon*. عندما تصبح اللغة ذات جذور عميقة ومؤسسة بشكل جيد ومتين مثل لغة C، فإن الاحتياجات الجديدة تكون في تطوير هذه اللغة وليس إزاحتها واستبدالها بلغة جديدة. طُوّرت لغة $C++$ من قبل العالم *Bjarne Stroustrup* في مختبرات بل *Bell laboratories* وسُمّيت بالأساس لغة C مع الأصناف *C with classes*. يتضمن الاسم، $C++$، عامل الزيادة (++) من لغة C ليشير إلى أن لغة $C++$ هي إصدار محسن من لغة C. تشمل لغة $C++$ لغة C؛ ولذا يمكن للمبرمجين استخدام مترجم $C++$ لترجمة برامج لغة C.

1.6.5 برنامج بسيط: جمع عددين صحيحين:

A Simple Program: Adding Tow Integers

يقوم البرنامج البسيط التالي بجمع عددين صحيحين، وهو يُوضح خصائص عدة للغة $C++$ مقارنة مع لغة C.

```
1.
2.   // Addition program
3.   #include <iostream>
4.
5.   int main()
6.   {
7.      int integer1;
8.
9.      std::cout << "Enter first integer\n";
10.     std::cin >> integer1;
11.
12.     int integer2, sum;          // declaration
13.
14.     std::cout << "Enter second integer\n";
15.     std::cin >> integer2;
16.     sum = integer1 + integer2;
17.     std::cout << "Sum is " << sum << std::endl;
```

```
18.
19.     return 0;   // indicate that program ended successfully
20.  } // end function main
```

خرج البرنامج

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

برنامج جمع عددين صحيحين

يبدأ السطر 2

// Addition program

بـ // لتوضيح أنّ هذا السطر عبارة عن تعليق *comment*. تسمح لنا هذه اللغة ببدء التعليق بـ // واستخدام باقي السطر لكتابة التعليق نفسه. كما يمكن استخدام أسلوب *C* في كتابة التعليق وهو وضع */ في بداية السطر و/* في نهاية السطر.

يُبيّن مُوجه المعالج الأولي للغة *C++*، *C++ preprocessor directive* (السطر 3)

#include <iostream>

أسلوب *C++* المعياري المعتمد من قِبل *ANSI/ISO* لتضمين ملفات الترويسة *header files* من المكتبة المعيارية *standard library*. يُخبر هذا السطر معالج *C++* الأولي بتضمين محتويات ملف ترويسة تدفق الإدخال/الإخراج *input/output stream header file* المسمّى *iostream*. يجب تضمين هذا الملف من أجل أيّ برنامج يريد إخراج البيانات على الشاشة أو إدخالها من لوحة المفاتيح.

السطر 5 وكما هو الحال بالنسبة للغة *C* هو جزء أساسي من اللغة *C++*، تُبيّن الكلمة المفتاحية *int* على يسار الدالة *main* أنها ترجع قيمة صحيحة. لاحظ أن المبرمج في لغة *C* ليس بحاجة لتحديد نوع قيمة الرجوع *return-value-type* للدوال *functions* بينما لغة *C++* بحاجة لتحديد قيمة الرجوع لجميع الدوال و إلا فسيولد المترجم خطأ قواعدياً *syntax error*.

السطر 7 هو تصريح عن متغير مألوف، وبشكل مغاير للغة *C* حيث يجب التصريح عن المتغيرات داخل الصندوق *block* (مجموعة الأقواس{}) قبل أيّ جملة قابلة للتنفيذ، فإن

التصريح عن المتغيرات في لغة $C++$ يمكن أن يتم في أيّ مكان داخل الصندوق. وهذا مُبين في السطر 12 حيث تم التصريح عن المتغيرين *integer2* و*sum*.

ملحوظة:

- عند وضع التصريحات *declarations* في بداية الدالة فمن الأفضل فصل هذه التصريحات عن الجمل القابلة للتنفيذ في تلك الدالة بسطر فارغ ليتضح انتهاء التصريحات وبدء الجمل القابلة للتنفيذ.

تستخدم الجملة في السطر 9 تدفق الخرج المعياري *cout* والعامل >> (عامل الإدخال إلى التدفق *stream insertion operator* الذي يلفظ "ضع في *put to*") لإخراج سلسلة النص. يتم الإدخال والإخراج في لغة $C++$ كتدفقات *streams* من المحارف. لذلك، عند تنفيذ الجملة السابقة فإنها ترسل تدفق المحارف *"enter first integer"* إلى كائن تدفق الإخراج المعياري *standard output stream object* والذي هو *std::cout* والموصل عادة إلى الشاشة.

تستخدم الجملة في السطر 10 كائن تدفق الإدخال *input stream object* المسمّى *cin* وعامل الاستخلاص من التدفق *stream extraction operator* والذي هو >> للحصول على قيمة من لوحة المفاتيح. استخدام هذا العامل (>>) يجعل *std::cin* تأخذ المحرف المدخل من وحدة تدفق الإدخال المعيارية والتي هي عادة لوحة المفاتيح. نفضل لفظ الجملة في السطر 10 كالتالي:

"*std::cin gives a value in integer1*"

أو ببساطة نقول:

" std::cin gives in integer1"

عندما يُنفذ الحاسب الجملة السابقة فإنه ينتظر المستخدم كي يدخل قيمة المتغير *integer1* يقوم الحاسب بعدئذٍ بتحويل المحرف المدخل إلى عدد صحيح ويخصص هذه القيمة إلى المتغير *integer1*.

يطبع السطر 14 الكلمات *Enter second integer* على الشاشة ثم يشير إلى بداية السطر التالي.

تحث هذه الجملة المستخدم كي يجري حدثاً ما. حيث نحصل من السطر 15 على قيمة *integer2* من المستخدم.

يُظهر السطر 17 سلسلة المحرف *Sum is* متبوعة بالقيمة العددية للمتغيّر *sum* متبوعة بـ *std::endl*. (*endl* هو اختصار لـ *"end line"*) والذي يدعى معالِج التدفق *stream manipulator*.

يطبع المعالِج *std::endl* سطراً جديداً ثم يُنظف مخزن (ماسك) الإخراج *" flushes the output buffer"*. هذا يعني ببساطة أنه في بعض الأنظمة يتم تراكم الخرج في الآلة حتى يصبح كمية كافية كي تستحق الظهور على الشاشة. يرغم المعالِج *std::endl* ظهور أيّ محارف متراكمة.

لاحظ أننا وضعنا *std::* قبل *cout* و*cin* و*endl*. هذا ضروري عند استخدام موجّه المعالج الأولي *#include <iostream>*. يُحدد الرمز *std::cout* أننا استخدمنا اسماً *name*، وهو في هذه الحالة *cout* والذي ينتمي إلى "فضاء الاسم *name space" std*. فضاءات الاسم *name spaces* هي إحدى مزايا *C++* المتطوّرة. إذاً يجب أن نتذكر ببساطة إدخال *std::* قبل كتابة كل من *cout* و*cin* و*cerr* في برنامج ما. يمكن أن يكون ذلك مربكاً، لكننا سنقوم بإدخال جملة *using*، والتي ستسمح لنا تجنّب كتابة *std::* قبل كل استخدام لفضاء الاسم *std*.

لاحظ أنّ الجملة في السطر 17 تخرج لنا قيماً متعددة بأنواع مختلفة (مثل *string* و*double* و*int* ...الخ) يعرف عامل الإدخال إلى التدفق كيفية إخراج كل قطعة من البيانات، يُشار إلى استخدام عوامل الإدخال إلى التدفق المتعددة (<<) في جملة مفردة باسم عمليات إدخال تدفق متتالية أو متتابعة *concatenating, chaining or cascading stream insertion operations*.

لذلك فمن غير الضروري أن يكون لدينا جمل خرج متعددة لإخراج قطع متعددة من البيانات. يمكن تنفيذ العمليات الحسابية في جمل الإخراج، كما يمكننا تجميع الجمل في السطرين 16 و17 في الجملة:

std::cout << integer1+integer2<< std::endl;

وبذا نلغي الحاجة للمتغير *sum*.

للغة *C++* سمة قوية جداً وهي أنها تُمكن المستخدمين من إنشاء أنواع بياناتهم الخاصة. أيضاً يمكن أن تُعلّم *C++* كيفية إدخال وإخراج قيم أنواع البيانات الجديدة

هذه باستخدام العاملين << و>> (هذا ما يعرف بالتحميل الزائد للعامل *operator overloading*).

2.6.5 مكتبة C++ المعيارية: *C++ Standard Library*

تُبنى برامج لغة $C++$ بصندوقي بناء رئيسيين هما الدوال *functions* وأنواع البيانات المعرّفة من قبل المستخدم *user-defined data types* والمسمّاة الأصناف *classes*. يستفيد معظم مبرمجي $C++$ من المجموعات الغنية للأصناف والدوال الموجودة في مكتبة $C++$ المعيارية، ولذلك فإنّ هناك جزآن للدخول إلى عالم $C++$. الأول هو تعلّم لغة $C++$ نفسها، والثاني هو تعلم كيفية استخدام الأصناف والدوال في مكتبة $C++$ المعيارية.

تُقدّم مكتبات الصنف المعيارية بشكل عام من قبل بائعي المترجمات، وتقدّم العديد من مكتبات الصنف ذات الأغراض الخاصة من قبل بائعي البرامج المستقلين *independent software vendors (ISV)*.

ميزة إنشاء الدوال والأصناف الخاصة هي أننا نعرف آلية عملها بالضبط. وسيئتها هي أنها مُضيعة للزمن *time consuming* ومن الصعب صيانتها *difficult to maintain*.